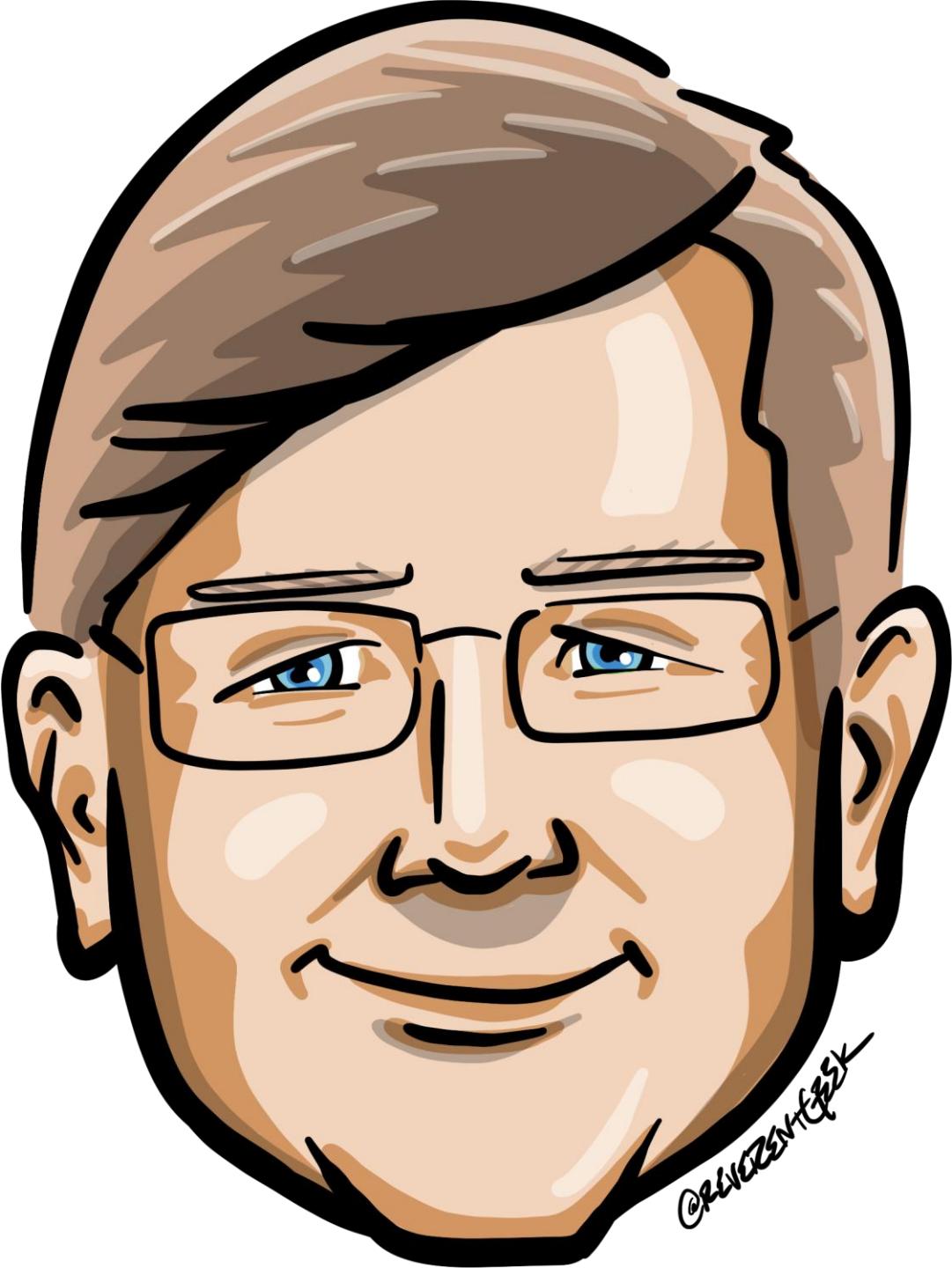




# Unlock the Power of Messaging Patterns

# Who is Chad Green?

- ✉ chadgreen@chadgreen.com
- ✳ TaleLearnCode
- 🌐 ChadGreen.com
- 🐦 ChadGreen & TaleLearnCode
- linkedin ChadwickEGreen



# Agenda

- 08:00 – Introduction
- 08:20 – Fundamentals of Messaging Patterns
- 08:40 – Lab Exercise: Publish/Subscribe Messaging
- 09:00 – Break
- 09:10 – Routing and Processing
- 09:30 – Lab Exercise: Dead Letter Queues
- 09:55 – Advanced Processing Techniques
- 10:15 – Lab Exercise: Claim Checks
- 10:35 – Break
- 10:45 – Resilience and Reliability
- 11:00 – Lab Exercise: Circuit Breaker
- 11:20 – Streaming Patterns
- 11:30 – Design Considerations
- 11:50 - Conclusion

# Objective & Learning Outcomes

## Objective

Equip attendees with the knowledge and skills to design, implement, and optimize message systems.

## Learning Outcomes

- Understand fundamentals and advanced messaging patterns.
- Apply best practices to real-world scenarios.
- Gain hands-on experience through practical exercises.
- Develop the ability to design resilient and efficient messaging systems

# Importance of Messaging Systems

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Why Messaging Systems Matter

Seamless  
Communications

Enable Asynchronous Communication

Integrate Diverse Technologies

Enhance Collaboration

# Why Messaging Systems Matter

Seamless  
Communications

Scalability and  
Performance

Load Distribution

Horizontal Scaling

Optimized Throughput

# Why Messaging Systems Matter

Seamless  
Communications

Scalability and  
Performance

Reliability and  
Fault Tolerance

Guaranteed Delivery

Persistent Storage

Error Handling

# Why Messaging Systems Matter

Seamless  
Communications

Scalability and  
Performance

Reliability and  
Fault Tolerance

Real-World  
Examples

E-Commerce

Financial Systems

IoT

# Key Concepts and Terminology

Message Queue

# Why Messaging Systems Matter

Message Queue

Publish/  
Subscriber Model

# Why Messaging Systems Matter

Message Queue

Publisher/  
Subscriber Model

Message Brokers

# Why Messaging Systems Matter

Message Queue

Publisher/  
Subscriber Model

Message Brokers

Transactions

# Why Messaging Systems Matter

Message Queue

Publisher/  
Subscriber Model

Message Brokers

Transactions

Dead Letter  
Queue

# Why Messaging Systems Matter

Message Queue

Publisher/  
Subscriber Model

Message Brokers

Transactions

Dead Letter  
Queue

Idempotence

# Why Messaging Systems Matter

Message Queue

Publisher/  
Subscriber Model

Message Brokers

Transactions

Dead Letter  
Queue

Idempotence

FIFO  
(First In, First Out)

# Fundamentals of Messaging



Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Overview of Messaging Patterns

Reliable and Scalable  
Communication

Help Decouple  
Components

# Lots of Messaging Patterns

## Basic Messaging Patterns

- Point-to-Point Messaging
- Publish/Subscribe Messaging
- Request/Reply
- Competing Consumers
- Dead Letter Queues

## Advanced Messaging Patterns

- Event Streaming
- Broadcast Pattern
- Aggregation Pattern
- Bidirectional Synchronization Pattern
- Correlation Pattern

## Design and Integration Patterns

- Choreography
- Orchestration

- Error Handling and Retry Policies
- Message Filter
- Recipient List
- Splitter
- Aggregator
- Resequencer
- Composed Message Processor
- Composed Message Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker
- Message Translator
- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check

- Normalizer
- Canonical Data Model
- Transactional Queues
- Saga
- Sequence Convoy
- Transactional Outbox
- Wire Tap
- Message History
- Message Store
- Smart Proxy
- Test Message Channel
- Purger
- Message Scheduler

## System Management Patterns

- Message Endpoint
- Messaging Gateway
- Durable Subscriber
- Idempotent Receiver
- Service Activator
- Polling Consumer
- Event-Driven Consumer
- Message Dispatcher
- Selective Consumer

## Other Key Concepts

- Message Routing
- Message Filtering
- Circuit Breaker
- Exactly Once Processing

# Lots of Messaging Patterns

## Basic Messaging Patterns

- Point-to-Point Messaging
- Publish/Subscribe Messaging
- Request/Reply
- Competing Consumers
- Dead Letter Queues

## Advanced Messaging Patterns

- Event Streaming
- Broadcast Pattern
- Aggregation Pattern
- Bidirectional Synchronization Pattern
- Correlation Pattern

## Design and Integration Patterns

- Choreography
- Orchestration

- Error Handling and Retry Policies
- Message Filter
- Recipient List
- Splitter
- Aggregator
- Resequencer
- Composed Message Processor
- Composed Message Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker
- Message Translator
- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check

- Normalizer
- Canonical Data Model
- Transactional Queues
- Saga
- Sequence Convoy
- Transactional Outbox
- Wire Tap
- Message History
- Message Store
- Smart Proxy
- Test Message Channel
- Purger
- Message Scheduler

## System Management Patterns

- Message Endpoint
- Messaging Gateway
- Durable Subscriber
- Idempotent Receiver
- Service Activator
- Polling Consumer
- Event-Driven Consumer
- Message Dispatcher
- Selective Consumer

## Other Key Concepts

- Message Routing
- Message Filtering
- Circuit Breaker
- Exactly Once Processing

# Lots of Messaging Patterns

## Basic Messaging Patterns

- Point-to-Point Messaging
- Publish/Subscribe Messaging
- Request/Reply
- Competing Consumers
- Dead Letter Queues

## Advanced Messaging Patterns

- Event Streaming
- Broadcast Pattern
- Aggregation Pattern
- Bidirectional Synchronization Pattern
- Correlation Pattern

## Design and Integration Patterns

- Choreography
- Orchestration

- Error Handling and Retry Policies
- Message Filter
- Recipient List
- Splitter
- Aggregator
- Resequencer
- Composed Message Processor
- Composed Message Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker
- Message Translator
- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check

- Normalizer
- Canonical Data Model
- Transactional Queues
- Saga
- Sequence Convoy
- Transactional Outbox

## System Management Patterns

- Message Endpoint
- Messaging Gateway
- Durable Subscriber
- Idempotent Receiver
- Service Activator
- Polling Consumer
- Event-Driven Consumer
- Message Dispatcher
- Selective Consumer

- Wire Tap
- Message History
- Message Store
- Smart Proxy
- Test Message Channel
- Purger
- Message Scheduler

## Other Key Concepts

- Message Routing
- Message Filtering
- Circuit Breaker
- Exactly Once Processing

# Importance of Understanding and Applying These Patterns

Improve Robustness  
and Efficiency

Handle Real-World  
Challenges

# Module Objectives

Understand the essential messaging patterns

Learn how to implement these patterns

Recognize the benefits and use cases for each pattern

# Point-to-Point Messaging

Fundamentals of Messaging

Unlock the Power of Messaging Patterns

# What is Point-to-Point Messaging?

Message sent from one sender to receiver

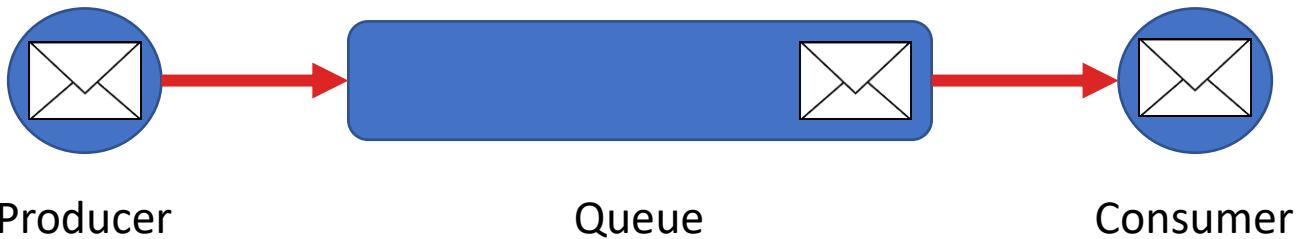
Straightforward, reliable way to send messages

# Key Components

**Message Producer  
(Sender)**

**Message Queue**

**Message Consumer  
(Receiver)**



# Benefits

Guaranteed  
Delivery

Simplicity

Decoupling

Load Balancing

Reliability

Flexibility

Control

# Drawbacks

Scalability Issues

Single Point of Failure

Resource Utilization

Message Ordering

Latency

Overhead

Error Handling Complexity

# Use Cases

Job Queue

Task Scheduling

Order Processing

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create a Queue

```
"Queues": [  
  {  
    "Name": "codemash.fundamentals.point-to-point",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLettereredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": false  
    }  
  }]
```

# Demonstration

Create Namespace

Create a Queue

Create a Message Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "codemash.fundamentals.point-to-point";

// Prompt the user to send a message
Console.WriteLine("Press any key to send a message to the queue...");
Console.ReadKey(true);

// Create a ServiceBusClient that will be used to create a sender
ServiceBusClient client = new(connectionString);

// Create a ServiceBusSender that we can use to send messages to the queue
ServiceBusSender sender = client.CreateSender(queueName);

// Create a message that we can send
ServiceBusMessage message = new("Hello, CodeMash!");

// Send the message
await sender.SendMessageAsync(message);
Console.WriteLine("==== Message Sent ====");
Console.WriteLine(message.Body.ToString());

// Close the sender and client
await sender.DisposeAsync();
await client.DisposeAsync();
```

# Demonstration

Create Namespace

Create a Queue

Create a Message Producer

Create a Message Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=HnDyJLcZGKQWzXqfCwvPjBkV0oIuR0EY";
const string queueName = "codemash.fundamentals.point-to-point";

async Task MessageHandler(ProcessMessageEventArgs args)
{
    Console.WriteLine();
    Console.WriteLine("==== Message Received ====");
    Console.WriteLine("Sequence Number: " + args.Message.SequenceNumber);
    Console.WriteLine("Enqueued Time: " + args.Message.EnqueueTime);
    Console.WriteLine("Message Body: " + args.Message.Body.ToString());
    await args.CompleteMessageAsync(args.Message);
}

Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine();
    Console.WriteLine("==== An error occurred ====");
    Console.WriteLine("Error Source: " + args.ErrorSource);
    Console.WriteLine("Exception: " + args.Exception.ToString());
    return Task.CompletedTask;
}

await processor.StopProcessingAsync();

// Close the processor and client
await processor.DisposeAsync();
await client.DisposeAsync();
```

# Demonstration

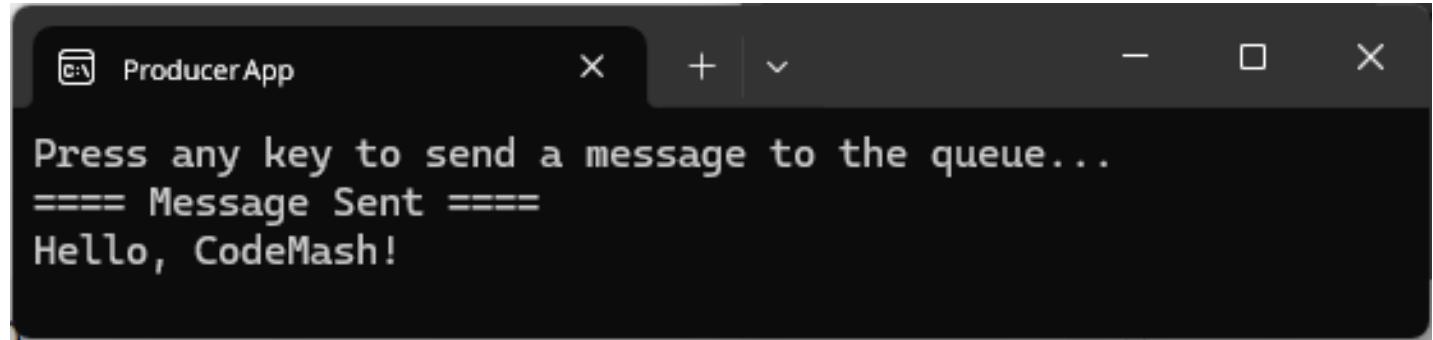
Create Namespace

Create a Queue

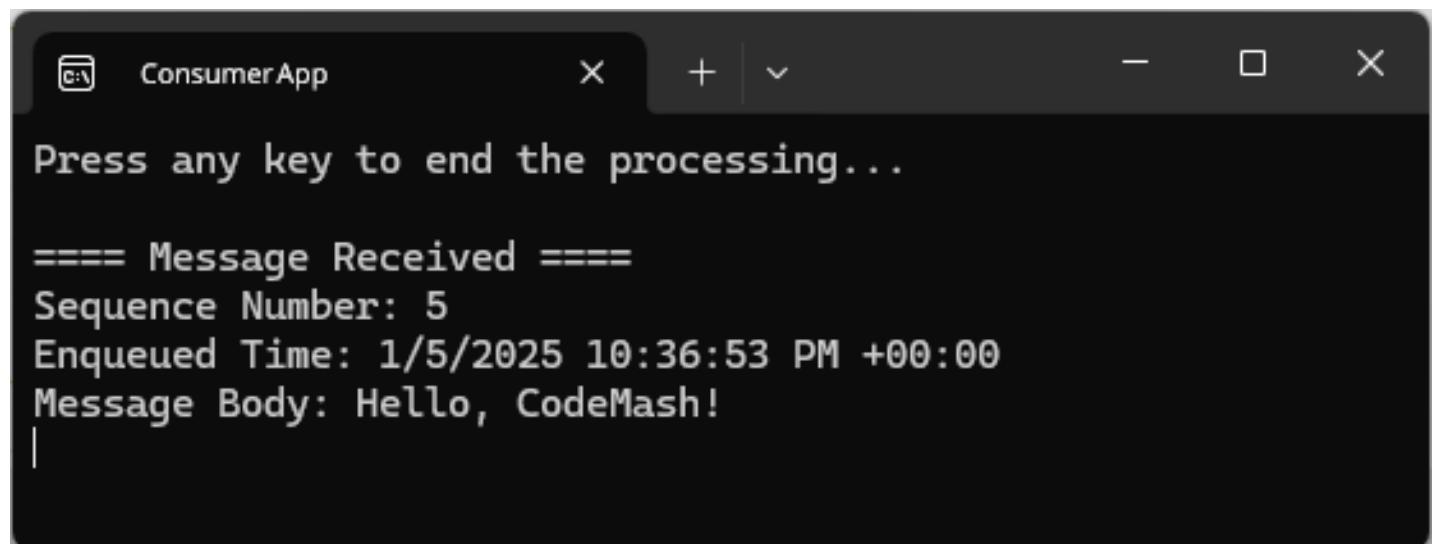
Create a Message Producer

Create a Message Consumer

Run the Demo



```
ProducerApp
Press any key to send a message to the queue...
==== Message Sent ====
Hello, CodeMash!
```



```
ConsumerApp
Press any key to end the processing...
==== Message Received ====
Sequence Number: 5
Enqueued Time: 1/5/2025 10:36:53 PM +00:00
Message Body: Hello, CodeMash!
|
```

# Key Points to Remember

- Ensures messages are processed by one consumer.
- Suitable for tasks that require guaranteed delivery to a single receiver.
- Helps in decoupling producers and consumers, allowing them to operate independently.

# Publish/Subscribe Messaging

Fundamentals of Messaging

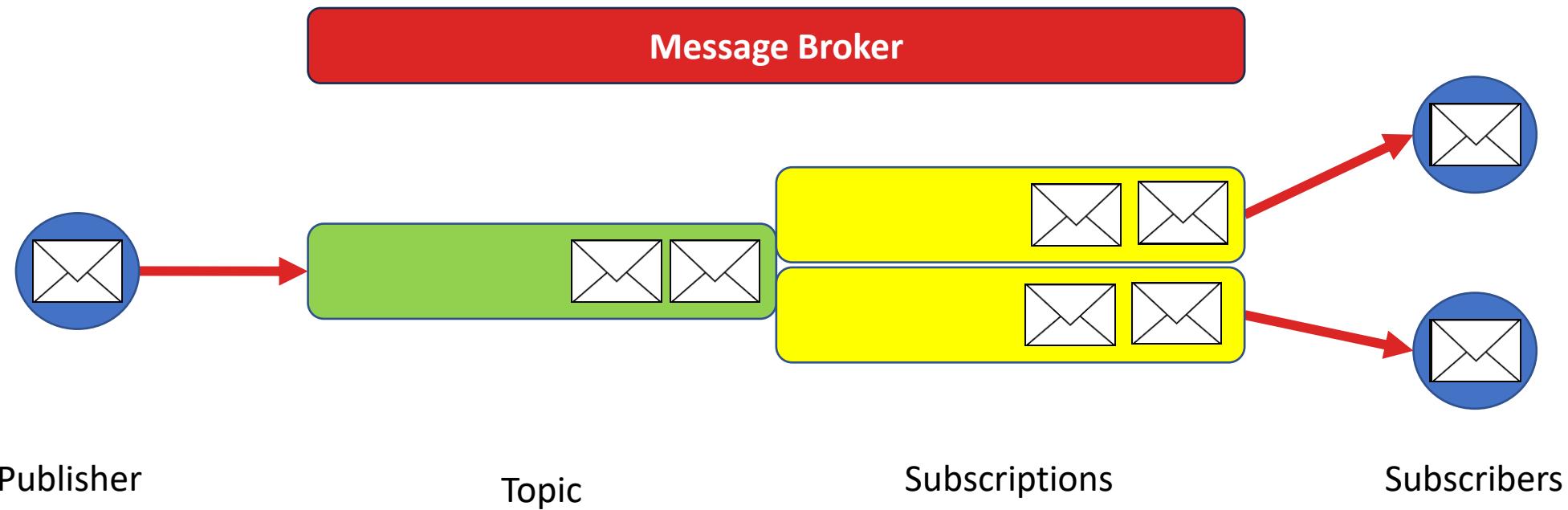
Unlock the Power of Messaging Patterns

# What is Pub/Sub Messaging?

Messages published to a topic for multiple subscribers

Enables scalable and flexible communication

# Pub/Sub Key Components & Flow



# Benefits

Decoupling

Multiple  
Consumers

Scalability &  
Flexibility

# Drawbacks

Complexity

# Drawbacks

Complexity

Scalability

# Drawbacks

Complexity

Scalability

Subscriber  
Dependency

# Drawbacks

Complexity

Scalability

Subscriber  
Dependency

Resource  
Utilization

# Drawbacks

Complexity

Scalability

Subscriber  
Dependency

Resource  
Utilization

Latency

# Drawbacks

Complexity

Scalability

Subscriber  
Dependency

Resource  
Utilization

Latency

Error Handling

# Use Cases

Broadcasting  
Events

Real-Time Data  
Feeds

Alert Systems

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create a Topic

```
"Topics": [  
  {  
    "Name": "codemash.fundamentals.publish-subscribe",  
    "Properties": {  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "RequiresDuplicateDetection": false  
    },  
    "Subscriptions": [  
    ]  
  }  
]
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

```
"Topics": [
    {
        "Name": "topic1",
        "Properties": {
            "DeadLetteringOnMessageExpiration": false,
            "DefaultMessageTimeToLive": "PT1H",
            "LockDuration": "PT1M",
            "MaxDeliveryCount": 3,
            "ForwardDeadLetteredMessagesTo": "",
            "ForwardTo": "",
            "RequiresSession": false
        }
    },
    {
        "Name": "topic2",
        "Properties": {
            "DeadLetteringOnMessageExpiration": false,
            "DefaultMessageTimeToLive": "PT1H",
            "LockDuration": "PT1M",
            "MaxDeliveryCount": 3,
            "ForwardDeadLetteredMessagesTo": "",
            "ForwardTo": "",
            "RequiresSession": false
        }
    }
],
"Subscriptions": [
    {
        "Name": "subscription1",
        "Properties": {
            "DeadLetteringOnMessageExpiration": false,
            "DefaultMessageTimeToLive": "PT1H",
            "LockDuration": "PT1M",
            "MaxDeliveryCount": 3,
            "ForwardDeadLetteredMessagesTo": "",
            "ForwardTo": "topic2",
            "RequiresSession": false
        }
    },
    {
        "Name": "subscription2",
        "Properties": {
            "DeadLetteringOnMessageExpiration": false,
            "DefaultMessageTimeToLive": "PT1H",
            "LockDuration": "PT1M",
            "MaxDeliveryCount": 3,
            "ForwardDeadLetteredMessagesTo": "",
            "ForwardTo": "topic1",
            "RequiresSession": false
        }
    }
]
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

Create Message Publisher

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string topicName = "codemash.fundamentals.publish-subscribe";

// Ask the user to press a key to send a message to the topic
Console.WriteLine("Press any key to send a message to the topic: codemash.fundamentals.publish-subscribe");
Console.ReadKey(true);

// Create a ServiceBusClient object that you can use to create a ServiceBusSender
await using ServiceBusClient client = new(connectionString);

// Create a ServiceBusSender object that you can use to send messages to the topic
ServiceBusSender sender = client.CreateSender(topicName);

// Create a message that we can send
ServiceBusMessage message = new("Hello, CodeMash Subscribers!");

// Send the message to the topic
await sender.SendMessageAsync(message);
Console.WriteLine($"Sending message: {message.Body}");

// Dispose of the client and sender
await sender.DisposeAsync();
await client.DisposeAsync();
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

Create Message Publisher

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string topicName = "codemash.fundamentals.publish-subscribe";

// Ask the user to press a key to send a message to the topic
Console.WriteLine("Press any key to send a message to the topic: codemash.fundamentals.publish-subscribe");
Console.ReadKey(true);

// Create a ServiceBusClient object that you can use to create a ServiceBusSender
await using ServiceBusClient client = new(connectionString);

// Create a ServiceBusSender object that you can use to send messages to the topic
ServiceBusSender sender = client.CreateSender(topicName);

// Create a message that we can send
ServiceBusMessage message = new("Hello, CodeMash Subscribers!");

// Send the message to the topic
await sender.SendMessageAsync(message);
Console.WriteLine($"Sending message: {message.Body}");

// Dispose of the client and sender
await sender.DisposeAsync();
await client.DisposeAsync();
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

Create Message Publisher

Create Subscriber Helper

```
public static async Task ProcessTopicSubscription(string connectionString, string topicName, string subscriptionName)
{
    // Create a ServiceBusClient object that you can use to create a ServiceBusProcessor
    await using ServiceBusClient client = new(connectionString);

    // Create a ServiceBusProcessor object that you can use to process messages from the subscription
    await using ServiceBusProcessor processor = client.CreateProcessor(topicName, subscriptionName);

    // Add an event handler to process messages
    processor.ProcessMessageAsync += args =>
    {
        string body = args.Message.Body.ToString();
        Console.WriteLine($"Received message: {body}");
        await args.CompleteMessageAsync(args.Message);
    };

    // Add an event handler to process any errors
    processor.ProcessErrorAsync += args =>
    {
        Console.WriteLine(args.Exception.ToString());
        return Task.CompletedTask;
    };

    // Start processing
    await processor.StartProcessingAsync();
    Console.WriteLine();
    Console.WriteLine("Press any key to stop processing messages...");
    Console.ReadKey(true);

    // Stop processing
    await processor.StopProcessingAsync();
}
}
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

Create Message Publisher

Create Subscriber Helper

Create Message Subscribers

## Consumer 1

```
using ConsumerLibrary;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string topicName = "codemash.fundamentals.publish-subscribe";
const string subscriptionName = "subscription1";

Console.WriteLine("Consumer 1 is listening for messages...");
await ServiceBusHelper.ProcessTopicSubscription(connectionString, topicName, subscriptionName);
```

## Consumer 2

```
using ConsumerLibrary;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string topicName = "codemash.fundamentals.publish-subscribe";
const string subscriptionName = "subscription2";

Console.WriteLine("Consumer 2 is listening for messages...");
await ServiceBusHelper.ProcessTopicSubscription(connectionString, topicName, subscriptionName);
```

# Demonstration

Create Namespace

Create a Topic

Create Subscriptions

Create Message Publisher

Create Subscriber Helper

Create Message Subscribers

Run the Demo

```
Producer x + ▾  
Press any key to send a message to the topic: codemash.fundamentals.publish-subscribe  
Sending message: Hello, CodeMash Subscribers!
```

```
Consumer1 x + ▾  
Consumer 1 is listening for messages...  
Press any key to stop processing messages...  
Received message: Hello, CodeMash Subscribers!  
|
```

```
Consumer2 x + ▾  
Consumer 2 is listening for messages...  
Press any key to stop processing messages...  
Received message: Hello, CodeMash Subscribers!  
|
```

# Key Points to Remember

- Supports one-to-many message delivery.
- Useful for broadcasting events or data to multiple consumers.
- Decouples publishers and subscribers, enhancing system flexibility.

# Competing Consumers

Fundamentals of Messaging

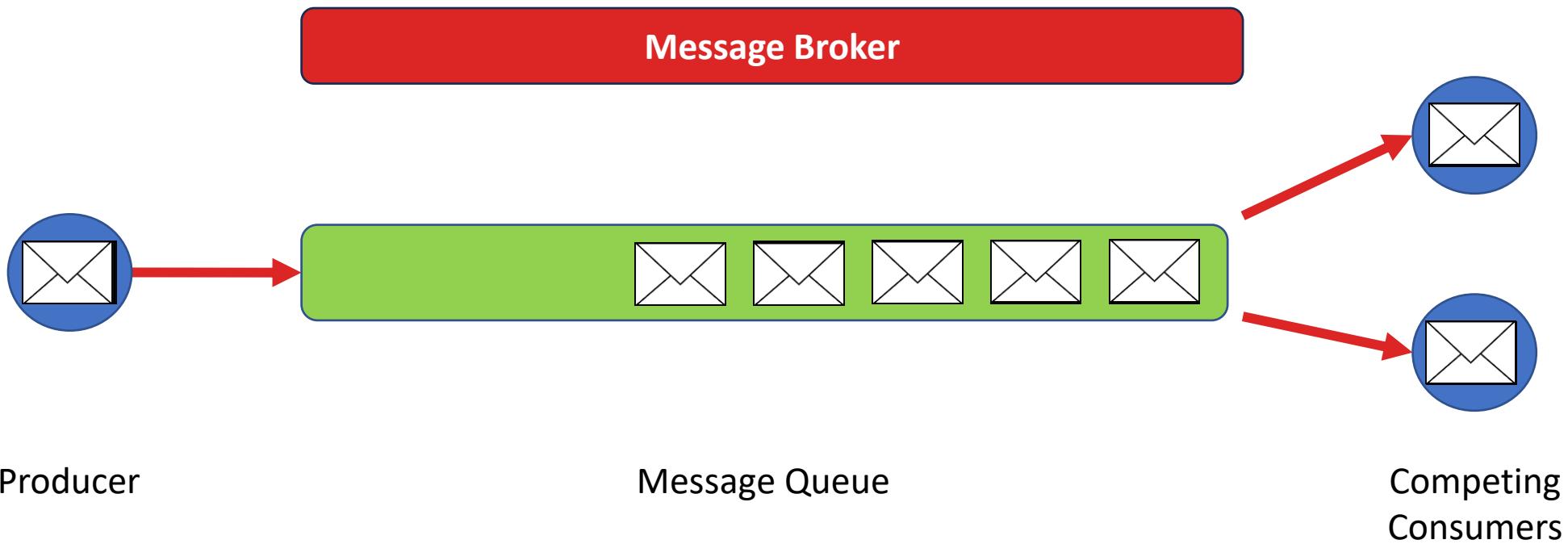
Unlock the Power of Messaging Patterns

# What is Competing Consumers

Multiple consumers read  
and process queue

Allows multiple  
consumers to process  
message concurrently

# Key Components & Flow



# Benefits

Scalability

Load Balancing

Fault Tolerance

# Drawbacks

Message  
Ordering

Complexity in  
Coordination

Resource  
Contention

# Use Cases

Task Processing

Data Processing

Order Processing

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create the Message Queue

```
"Queues": [  
  {  
    "Name": "codemash.competingconsumers",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLetteredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": false  
    }  
  }  
]
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessK
const string queueName = "codemash.competingconsumers.taskQueue";

// Wait for user input before sending messages
Console.WriteLine("Press any key to send messages to the queue...");
Console.ReadKey(true);

// Create a new Service Bus client
await using ServiceBusClient client = new(connectionString);

// Create a sender for the queue
await using ServiceBusSender sender = client.CreateSender(queueName);

// Send messages to the queue
Console.WriteLine();
Console.WriteLine("Sending messages to the queue...");
for (int i = 0; i < 10; i++)
{
    ServiceBusMessage message = new($"Message {i}");
    await sender.SendMessageAsync(message);
    Console.WriteLine($"Sent message: {message.Body}");
}
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

Create Competing Consumers

```
using Azure.Messaging.ServiceBus;  
const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";  
const string queueName = "codemash.competingconsumers.taskQueue";  
  
// Create a new Service Bus client  
await using var client = new ServiceBusClient(connectionString);  
  
// Create a Service Bus processor for the queue  
await using var processor = client.CreateProcessor(queueName);  
  
// Add a handler to process messages  
processor.ProcessMessageAsync += async args =>  
{  
    Console.WriteLine();  
    Console.WriteLine($"Consumer 2 processing message: {args.Message.Body}");  
    await Task.Delay(new Random().Next(200, 1500)); // Simulate variable processing time  
    await args.CompleteMessageAsync(args.Message);  
    Console.WriteLine($"Consumer 2 completed message: {args.Message.Body}");  
};  
  
// Add a handler to process any errors  
processor.ProcessErrorAsync += args =>  
{  
    Console.WriteLine($"Error: {args.Exception}");  
    return Task.CompletedTask;  
};  
  
// Start processing messages  
Console.WriteLine("Console 2 is ready to process messages; press any key to stop processing...");  
Console.ReadKey(true);  
  
// Stop processing messages  
await processor.StopProcessingAsync();  
Console.WriteLine("Stopped processing messages.");
```

# Demonstration

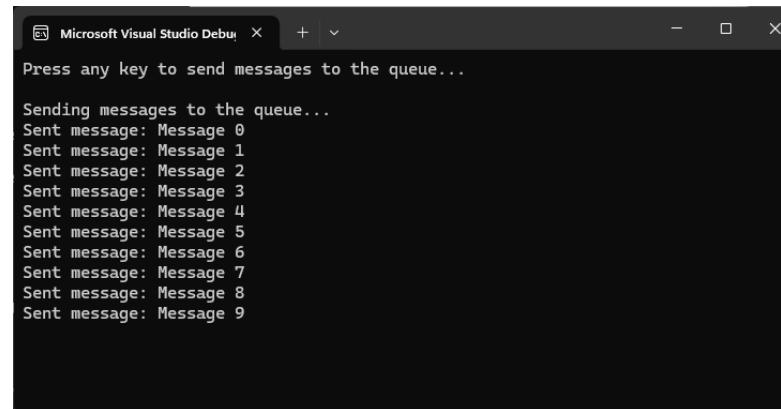
Create Namespace

Create the Message Queue

Create a Message Producer

Create Competing Consumers

Run the Demo

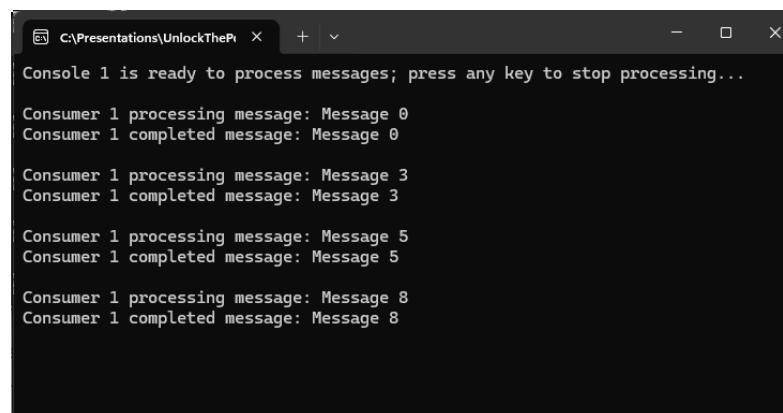


Microsoft Visual Studio Debug

Press any key to send messages to the queue...

Sending messages to the queue...

Sent message: Message 0  
Sent message: Message 1  
Sent message: Message 2  
Sent message: Message 3  
Sent message: Message 4  
Sent message: Message 5  
Sent message: Message 6  
Sent message: Message 7  
Sent message: Message 8  
Sent message: Message 9



C:\Presentations\UnlockTheP... X + v

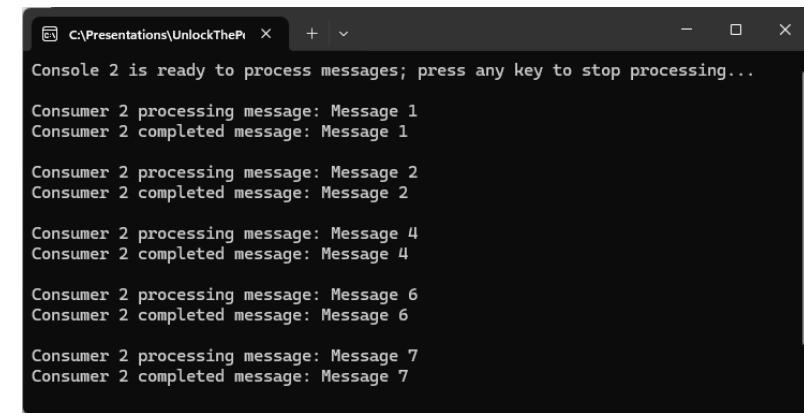
Console 1 is ready to process messages; press any key to stop processing...

Consumer 1 processing message: Message 0  
Consumer 1 completed message: Message 0

Consumer 1 processing message: Message 3  
Consumer 1 completed message: Message 3

Consumer 1 processing message: Message 5  
Consumer 1 completed message: Message 5

Consumer 1 processing message: Message 8  
Consumer 1 completed message: Message 8



C:\Presentations\UnlockTheP... X + v

Console 2 is ready to process messages; press any key to stop processing...

Consumer 2 processing message: Message 1  
Consumer 2 completed message: Message 1

Consumer 2 processing message: Message 2  
Consumer 2 completed message: Message 2

Consumer 2 processing message: Message 4  
Consumer 2 completed message: Message 4

Consumer 2 processing message: Message 6  
Consumer 2 completed message: Message 6

Consumer 2 processing message: Message 7  
Consumer 2 completed message: Message 7

# Key Points to Remember

- Supports concurrent message processing by multiple consumers.
- Enhances system scalability, throughput, and reliability.
- Effective for load balancing and fault tolerance.

# Request/Reply Messaging

Fundamentals of Messaging

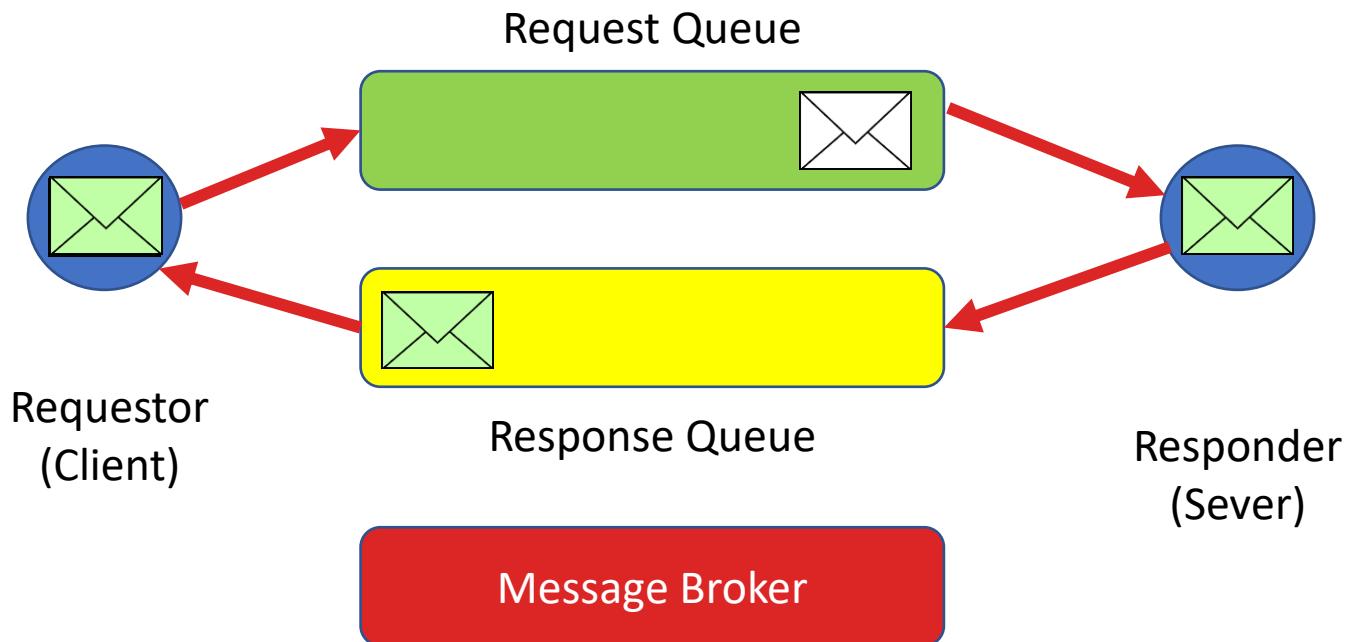
Unlock the Power of Messaging Patterns

# What is Request/Reply Messaging?

Requester waits for a  
reply message

Enables synchronous  
communication

# Key Components & Flow



# Benefits

Synchronous  
Communication

Simplified  
Workflows

Reliability

Control and  
Coordination

Decoupling

# Drawbacks

Scalability

Latency

Resource  
Utilization

Complexity in  
Error Handling

Single Point of  
Failure

# Use Cases

Service  
Invocation

Data Retrieval

Status Updates

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create the Request Queue

```
"Queues": [
  {
    "Name": "codemash.requestreply.request",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

# Demonstration

Create Namespace

Create the Request Queue

Create the Response Queue

```
"Queues": [  
  {  
    "Name": "codemash.requestreply.response",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLetteredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": false  
    }  
  }  
]
```

# Demonstration

Create Namespace

Create the Request Queue

Create the Response Queue

Create the Requestor

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=Roo
const string requestQueueName = "codemash.requestreply.request";
const string responseQueueName = "codemash.requestreply.response";

// Prompt the user to press a key to send a request message
Console.WriteLine("Press any key to send a request message...");
Console.ReadKey();

// Create a ServiceBusClient that you can use to create a ServiceBusSender
await using ServiceBusClient client = new(connectionString);

// Create a sender for the request queue
ServiceBusSender requestSender = client.CreateSender(requestQueueName);

// Create a receiver for the response queue
ServiceBusReceiver responseReceiver = client.CreateReceiver(responseQueueName);

// Create a message that we will send to the request queue
ServiceBusMessage requestMessage = new("Request: What is the time?")
{
    ReplyTo = responseQueueName,
    CorrelationId = Guid.NewGuid().ToString()
};
```

# Demonstration

Create Namespace

Create the Request Queue

Create the Response Queue

Create the Requestor

```
// Send the message to the request queue
await requestSender.SendMessageAsync(requestMessage);
Console.WriteLine();
Console.WriteLine("==== Request Sent ====");
Console.WriteLine($"CorrelationId: {requestMessage.CorrelationId}");
Console.WriteLine($"ReplyTo: {requestMessage.ReplyTo}");
Console.WriteLine($"Body: {requestMessage.Body}");

// Receive the response from the response queue
ServiceBusReceivedMessage responseMessage = await responseReceiver.ReceiveMessageAsync();
Console.WriteLine();
Console.WriteLine("==== Response Received ====");
Console.WriteLine($"CorrelationId: {responseMessage.CorrelationId}");
Console.WriteLine($"Body: {responseMessage.Body}");
```

# Demonstration

Create Namespace

```
const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string requestQueueName = "codemash.requestreply.request";
```

Create the Request Queue

```
// Create a ServiceBusClient that you can use to create a ServiceBusReceiver
await using ServiceBusClient client = new(connectionString);
```

Create the Response Queue

```
// Create a processor for the request queue
ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(requestQueueName);
```

Create the Requestor

Create the Responder

# Demonstration

Create Namespace

Create the Request Queue

Create the Response Queue

Create the Requestor

Create the Responder

```
// Register a handler for processing messages
serviceBusProcessor.ProcessMessageAsync += async args =>

    ServiceBusReceivedMessage requestMessage = args.Message;
    Console.WriteLine();
    Console.WriteLine("===== Request Received =====");
    Console.WriteLine($"CorrelationId: {requestMessage.CorrelationId}");
    Console.WriteLine($"ReplyTo: {requestMessage.ReplyTo}");
    Console.WriteLine($"Body: {requestMessage.Body}");

    ServiceBusMessage responseMessage = new("The time is " + DateTime.Now)
    {
        CorrelationId = requestMessage.CorrelationId
    };

    string replyTo = requestMessage.ReplyTo;
    if (!string.IsNullOrEmpty(replyTo))
    {
        await using ServiceBusSender responseSender = client.CreateSender(replyTo);
        await responseSender.SendMessageAsync(responseMessage);
        Console.WriteLine();
        Console.WriteLine("===== Response Sent =====");
        Console.WriteLine($"CorrelationId: {responseMessage.CorrelationId}");
        Console.WriteLine($"Body: {responseMessage.Body}");
    }

    await args.CompleteMessageAsync(requestMessage);
};
```

# Demonstration

Create Namespace

Create the Request Queue

Create the Response Queue

Create the Requestor

Create the Responder

```
// Register a handler for any errors that occur when processing messages
serviceBusProcessor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};
```

# Demonstration

Create Namespace

```
// Start processing
await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine($"Listening for messages on: {requestQueueName}. Press any key to exit.");
Console.ReadKey();
```

Create the Request Queue

```
// Stop processing
await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine();
Console.WriteLine("Stopped processing messages.");
```

Create the Response Queue

Create the Requestor

Create the Responder

# Demonstration

Create Namespace

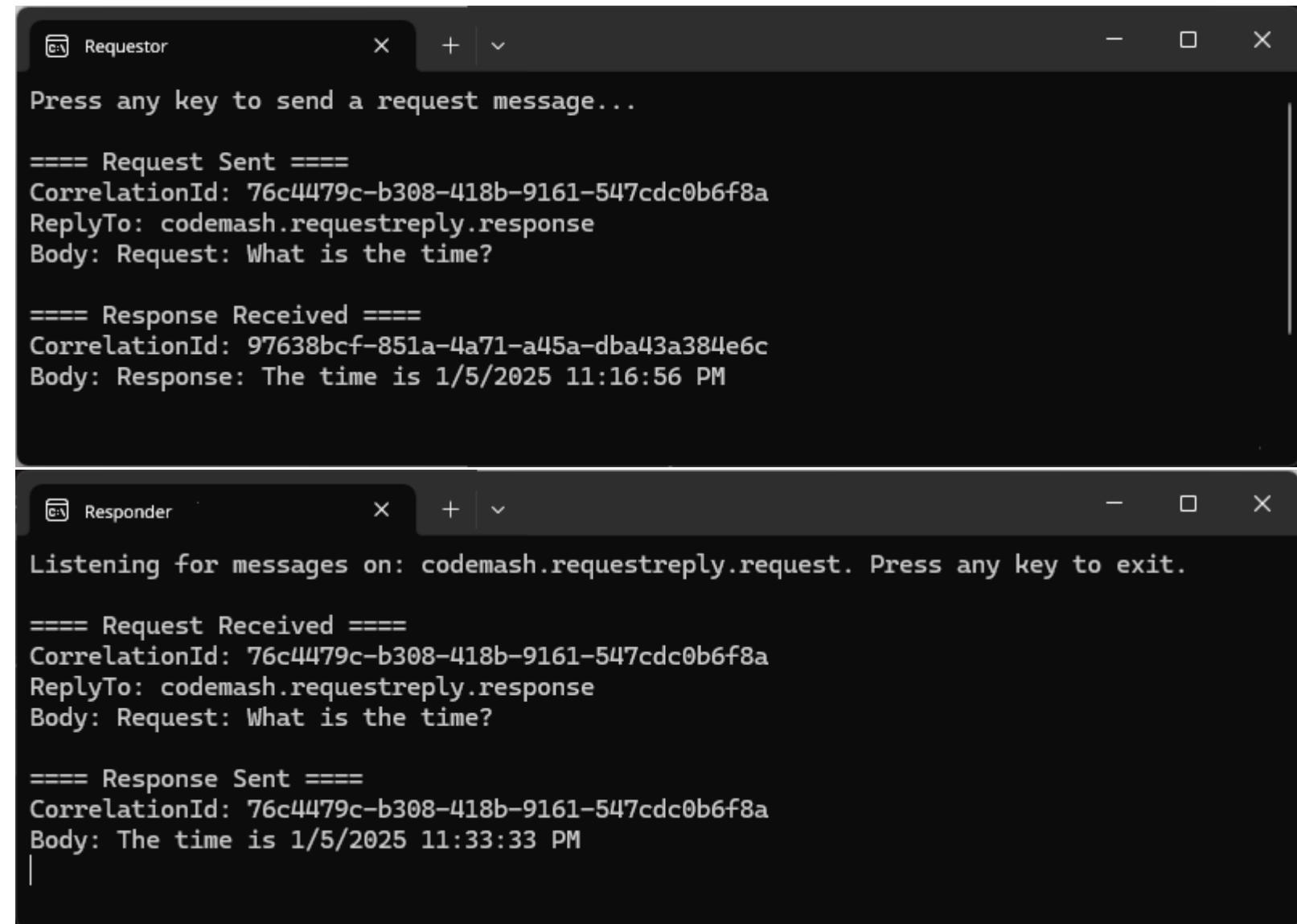
Create the Request Queue

Create the Response Queue

Create the Requestor

Create the Responder

Run the Demo



The image shows two terminal windows side-by-side. The left window is titled 'Requestor' and the right window is titled 'Responder'. Both windows have a dark theme with light-colored text.

**Requestor Window:**

```
Requestor
x + v
Press any key to send a request message...
===== Request Sent =====
CorrelationId: 76c4479c-b308-418b-9161-547cdc0b6f8a
ReplyTo: codemash.requestreply.response
Body: Request: What is the time?

===== Response Received =====
CorrelationId: 97638bcf-851a-4a71-a45a-dba43a384e6c
Body: Response: The time is 1/5/2025 11:16:56 PM
```

**Responder Window:**

```
Responder
x + v
Listening for messages on: codemash.requestreply.request. Press any key to exit.

===== Request Received =====
CorrelationId: 76c4479c-b308-418b-9161-547cdc0b6f8a
ReplyTo: codemash.requestreply.response
Body: Request: What is the time?

===== Response Sent =====
CorrelationId: 76c4479c-b308-418b-9161-547cdc0b6f8a
Body: Response: The time is 1/5/2025 11:33:33 PM
```

# Key Points to Remember

- Supports two-way communication between services or components.
- Suitable for scenarios requiring immediate response or confirmation.
- Simplifies synchronous interactions and workflows.

# Wrap-Up

Fundamentals of Messaging

Unlock the Power of Messaging Patterns

# When to Use One Over the Others

Dead Letter  
Queue

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Dead Letter  
Queue

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# When to Use One Over the Others

Point-to-Point

Publish/  
Subscribe

Competing  
Consumers

Request/Reply

# Importance in Building Robust Cloud Applications

Reliability

Scalability

Decoupling

Fault Tolerance

Flexibility

# Preview of What's Next in the Workshop

Routing and Processing

Advanced Processing Techniques

Resilience and Reliability

Steaming and Integrations

Design Considerations

Hands-On Exercises

# Preview of Routing and Processing

Message Routing

Dead Letter  
Queues

Message Filtering

Aggregator

Scatter-Gather

# Routing and Processing

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Message Routing

Routing and Processing

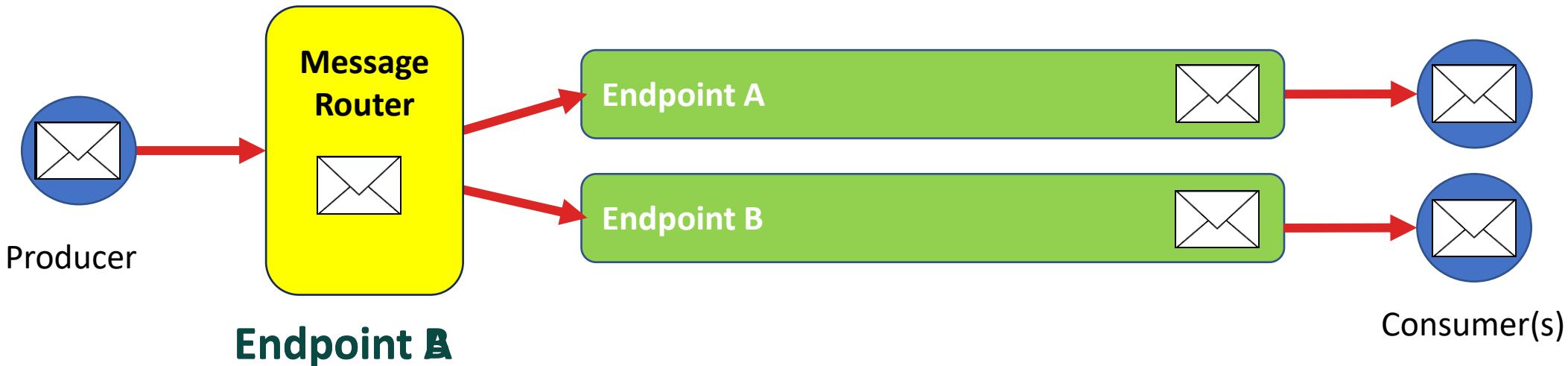
Unlock the Power of Messaging Patterns

# What is Messaging Routing

Directs messages to different destinations based on specific criteria

Isolate problematic messages

# Key Components & Flow



# Key Components & Flow



# Benefits

Flexibility

Scalability

Organization

# Drawbacks

Complexity

Overhead

Scalability

# Use Cases

Load Balancing

Content-Based  
Routing

Priority Routing

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create the Message Queues

```
"Queues": [
  {
    "Name": "codemash.routing.low-priority",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "codemash.routing.high-priority",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string highPriorityQueue = "codemash.routing.high-priority";
const string lowPriorityQueue = "codemash.routing.low-priority";

for (int i = 0; i < 10; i++)
{
    // Determine the priority of the message
    string priority = (i % 3 == 0) ? "high" : "low";

    // Create a new message
    ServiceBusMessage message = new($"Message {i + 1}");
    message.ApplicationProperties.Add("priority", priority);

    // Send the message to the appropriate queue based on its priority
    if (priority == "high")
        await highPrioritySender.SendMessageAsync(message);
    else
        await lowPrioritySender.SendMessageAsync(message);

    Console.WriteLine($"\\tSent message {i + 1} with priority {priority}");

}

// Wait for the user to press a key before exiting
Console.WriteLine();
Console.WriteLine("Finished sending messages. Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

Create a Message Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccess";
const string highPriorityQueue = "codemash.routing.high-priority";
const string lowPriorityQueue = "codemash.routing.low-priority";

// Process messages from the high-priority queue
await ProcessMessagesAsync(highPriorityQueue, "high");

// Process messages from the low-priority queue
await ProcessMessagesAsync(lowPriorityQueue, "low");

2 references | - changes | -authors, -changes
async Task ProcessMessagesAsync(string queueName, string priority)
{
    // Create a new instance of the ServiceBusClient
    await using ServiceBusClient client = new(connectionString);

    // Create a new instance of the ServiceBusProcessor for the specified queue
    await using ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(queueName);

    // Register a message handler for the queue
    serviceBusProcessor.ProcessMessageAsync += async args =>
    {
        // Process the message
        Console.WriteLine($"Received {priority}-priority message: {args.Message.Body}");
        await args.CompleteMessageAsync(args.Message);
    };

    // Register an error handler for the queue
    serviceBusProcessor.ProcessErrorAsync += args =>
    {
        Console.WriteLine($"Error processing {priority}-priority message: {args.Exception.Message}");
        return Task.CompletedTask;
    };

    // Start processing messages from the queue
    await serviceBusProcessor.StartProcessingAsync();
    Console.WriteLine();
    Console.WriteLine($"Press any key to stop processing {priority} priority messages...");
    Console.ReadKey(true);
};

// Stop processing messages from the queue
await serviceBusProcessor.StopProcessingAsync();
}
```

Unlock the Power of Messaging Patterns

# Demonstration

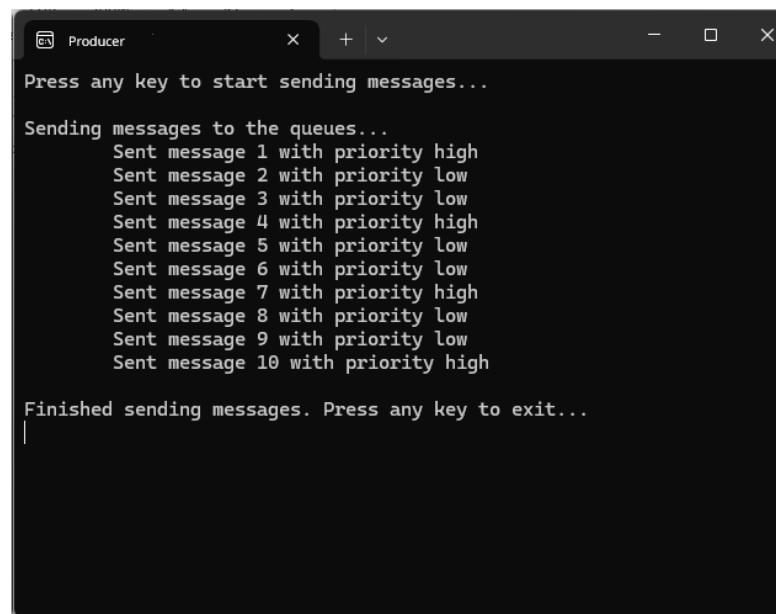
Create Namespace

Create the Message Queue

Create a Message Producer

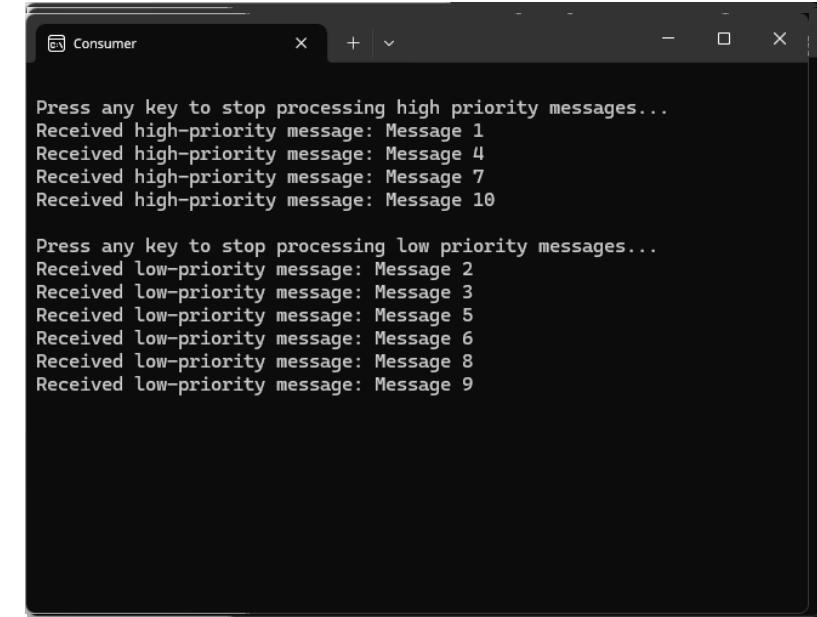
Create Competing Consumers

Run the Demo



Producer

```
Press any key to start sending messages...
Sending messages to the queues...
Sent message 1 with priority high
Sent message 2 with priority low
Sent message 3 with priority low
Sent message 4 with priority high
Sent message 5 with priority low
Sent message 6 with priority low
Sent message 7 with priority high
Sent message 8 with priority low
Sent message 9 with priority low
Sent message 10 with priority high
Finished sending messages. Press any key to exit...
```



Consumer

```
Press any key to stop processing high priority messages...
Received high-priority message: Message 1
Received high-priority message: Message 4
Received high-priority message: Message 7
Received high-priority message: Message 10
Press any key to stop processing low priority messages...
Received low-priority message: Message 2
Received low-priority message: Message 3
Received low-priority message: Message 5
Received low-priority message: Message 6
Received low-priority message: Message 8
Received low-priority message: Message 9
```

# Key Points to Remember

- Supports flexible and dynamic routing of messages.
- Enhances system scalability and load balancing.
- Facilitates efficient organization and management of message flows.

# Dead Letter Queues

Routing and Processing

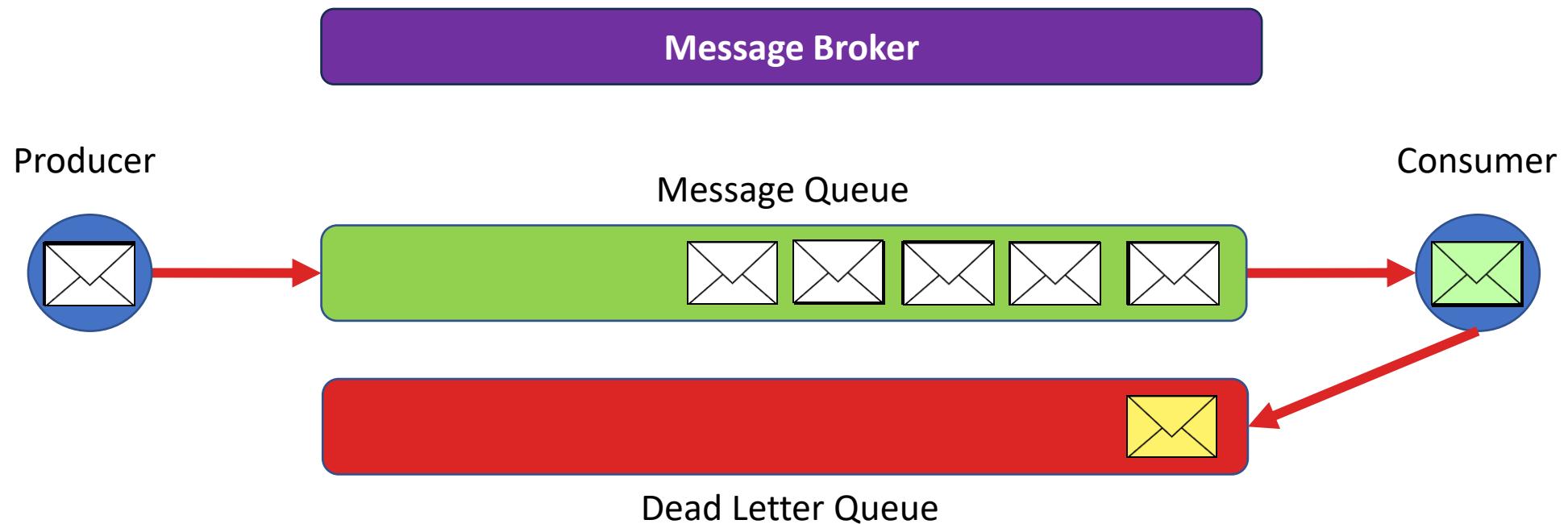
Unlock the Power of Messaging Patterns

# What are Dead Letter Queues

Secondary queue of  
unprocessable messages

Isolate problematic  
messages

# Key Components & Flow



# Benefits

Reliability

Troubleshooting

Efficiency

# Drawbacks?

Complexity

Delayed  
Processing

Data Loss  
Potential

# Use Cases

Error Handling

Message Expiry

Monitoring &  
Troubleshooting

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create the Message Queue

```
{  
  "Queues": [  
    {  
      "Name": "codemash.deadletter",  
      "Properties": {  
        "DeadLetteringOnMessageExpiration": false,  
        "DefaultMessageTimeToLive": "PT1H",  
        "DuplicateDetectionHistoryTimeWindow": "PT20S",  
        "ForwardDeadLetteredExceptionsTo": "",  
        "ForwardTo": "",  
        "LockDuration": "PT1M",  
        "MaxDeliveryCount": 3,  
        "RequiresDuplicateDetection": false,  
        "RequiresSession": false  
      }  
    }  
  ]  
}
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKey";
const string queueName = "codemash.deadletter";

// Wait for user input to start sending messages
Console.WriteLine("Press any key to send messages to the queue...");
Console.ReadKey(true);

// Create a Service Bus client
await using ServiceBusClient client = new(connectionString);

// Create a sender for the queue
await using ServiceBusSender sender = client.CreateSender(queueName);

// Create messages to send to the queue
Console.WriteLine();
Console.WriteLine("Sending messages to the queue...");
for (int i = 0; i < 10; i++)
{
    string messageContent = i % 2 == 0 ? $"Task {i}" : $"Task {i} fail";
    ServiceBusMessage message = new(messageContent);
    await sender.SendMessageAsync(message);
    Console.WriteLine($"\\tSent message: {messageContent}");
}
Console.WriteLine("All messages sent to the queue.");
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

Create a Message Consumer

```
// Add handler to process messages
processor.ProcessMessageAsync += async args =>
{
    try
    {
        Console.WriteLine($"Processing message: {args.Message.Body}");
        if (args.Message.Body.ToString().Contains("fail"))
            throw new Exception("Simulated exception");
        await args.CompleteMessageAsync(args.Message);
        Console.WriteLine("\tMessage processed successfully.");
    }
    catch (Exception ex)
    {

        Console.BackgroundColor = ConsoleColor.Red;
        Console.ForegroundColor = ConsoleColor.White;
        Console.WriteLine($" \tError processing message: {ex.Message}");
        Console.ResetColor();

        Dictionary<string, object> deadLetterProperties = new()
        {
            {"DeadLetterReason", "Processing error"},
            {"DeadLetterErrorMessage", ex.Message }
        };
        await args.DeadLetterMessageAsync(args.Message, deadLetterProperties);
    }
};
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

Create a Message Consumer

Create a DLQ Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=XXXXXXXXXX";
const string queueName = "codemash.deadletter/$DeadLetterQueue";

Console.WriteLine("Dead Letter Consumer");

// Create a Service Bus client
await using ServiceBusClient client = new(connectionString);

// Add handler to process messages
processor.ProcessMessageAsync += async args =>
{
    Console.WriteLine($"Processing DLQ message: {args.Message.Body}");
    if (args.Message.ApplicationProperties.TryGetValue("DeadLetterReason", out object? reason))
    {
        Console.WriteLine($"Dead Letter Reason: {reason}");
    }
    if (args.Message.ApplicationProperties.TryGetValue("DeadLetterErrorMessage", out object? errorMessage))
    {
        Console.WriteLine($"Dead Letter Error Message: {errorMessage}");
    }
    await args.CompleteMessageAsync(args.Message);
};

// Add handler to process any errors
processor.ProcessErrorAsync += args =>
{
    Console.WriteLine($"Error: {args.Exception.Message}");
    return Task.CompletedTask;
};

// Start processing
await processor.StartProcessingAsync();
Console.WriteLine("Processing messages from the dead letter queue. Press any key to exit...");
Console.ReadKey(true);
Console.WriteLine();

// Stop processing
await processor.StopProcessingAsync();
Console.WriteLine("Stopped processing messages.");
}
```

# Demonstration

Create Namespace

Create the Message Queue

Create a Message Producer

Create Competing Consumers

Create a DLQ Consumer

Run the Demo

```
Producer x + - □ ×
Press any key to send messages to the queue...
Sending messages to the queue...
Sent message: Task 0
Sent message: Task 1 fail
Sent message: Task 2
Sent message: Task 3 fail
Sent message: Task 4
Sent message: Task 5 fail
Sent message: Task 6
Sent message: Task 7 fail
Sent message: Task 8
Sent message: Task 9 fail
All messages sent to the queue.
```

```
Consumer x + - □ ×
Consumer
Processing messages from the queue. Press any key to exit...
Processing message: Task 0
    Message processed successfully.
Processing message: Task 1 fail
    Error processing message: Simulated exception
Processing message: Task 2
    Message processed successfully.
Processing message: Task 3 fail
    Error processing message: Simulated exception
Processing message: Task 4
    Message processed successfully.
Processing message: Task 5 fail
    Error processing message: Simulated exception
Processing message: Task 6
    Message processed successfully.
Processing message: Task 7 fail
    Error processing message: Simulated exception
Processing message: Task 8
    Message processed successfully.
Processing message: Task 9 fail
    Error processing message: Simulated exception
```

```
Dead Letter Queue Consumer x + - □ ×
Dead Letter Consumer
Processing messages from the dead letter queue. Press any key to exit...
Processing DLQ message: Task 1 fail
    Dead Letter Reason: Processing error
    Dead Letter Error Message: Simulated exception
Processing DLQ message: Task 3 fail
    Dead Letter Reason: Processing error
    Dead Letter Error Message: Simulated exception
Processing DLQ message: Task 5 fail
    Dead Letter Reason: Processing error
    Dead Letter Error Message: Simulated exception
Processing DLQ message: Task 7 fail
    Dead Letter Reason: Processing error
    Dead Letter Error Message: Simulated exception
Processing DLQ message: Task 9 fail
    Dead Letter Reason: Processing error
    Dead Letter Error Message: Simulated exception
```

# Key Points to Remember

- Helps in isolating and handling problematic messages.
- Facilitates troubleshooting and improves system reliability.
- Ensures that normal message processing continues without disruption.

# Message Filtering

Routing and Processing

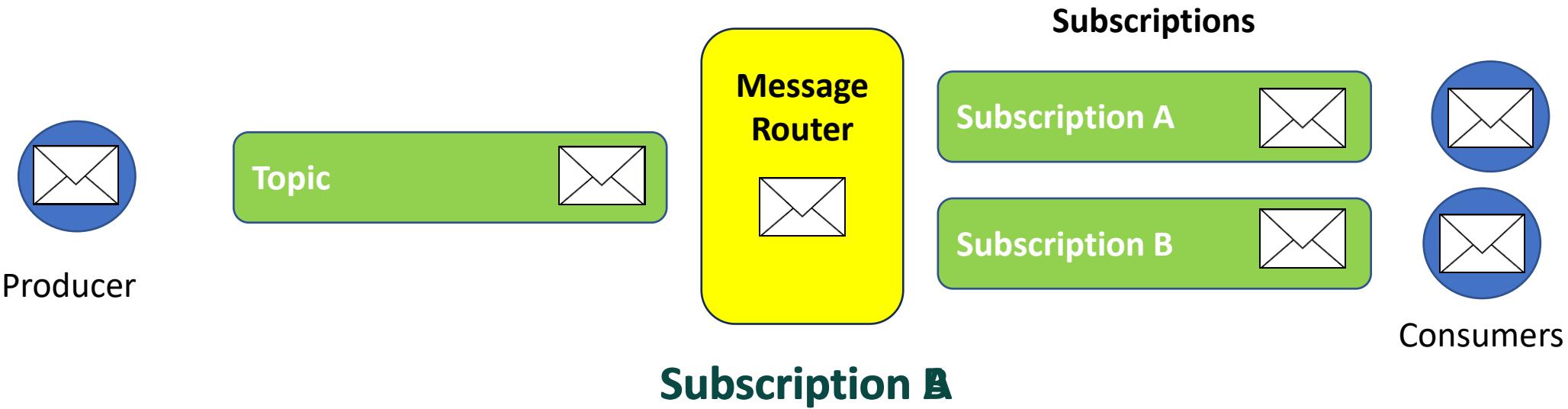
Unlock the Power of Messaging Patterns

# What is Messaging Filtering

Consumers selectively receive messages based on criteria or filters.

Isolate problematic messages

# Key Components & Flow



# Benefits

Efficiency

Scalability

Flexibility

# Drawbacks

Complexity

Overhead

Scalability

# Use Cases

Subscription  
Filtering

Data  
Segmentation

Notification  
Systems

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

```
"Subscriptions": [
    {
        "Name": "HighPriority",
        "Properties": {
            "DeadLetteringOnMessageExpiration": false,
            "DefaultMessageTimeToLive": "PT1H",
            "LockDuration": "PT1M",
            "MaxDeliveryCount": 3,
            "ForwardDeadLetteredMessagesTo": "",
            "ForwardTo": "",
            "RequiresSession": false
        },
        "Rules": [
            {
                "Name": "app-prop-filter-1",
                "Properties": {
                    "FilterType": "Correlation",
                    "CorrelationFilter": {
                        "Label": "High"
                    }
                }
            }
        ],
        {
            "Name": "LowPriority",
            "Properties": {
                "DeadLetteringOnMessageExpiration": true,
                "DefaultMessageTimeToLive": "P1D",
                "LockDuration": "PT1M",
                "MaxDeliveryCount": 5,
                "ForwardDeadLetteredMessagesTo": "deadletterqueue",
                "ForwardTo": "deadletterqueue",
                "RequiresSession": false
            },
            "Rules": [
                {
                    "Name": "app-prop-filter-2",
                    "Properties": {
                        "FilterType": "Correlation",
                        "CorrelationFilter": {
                            "Label": "Low"
                        }
                    }
                }
            ]
        }
    }
]
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create a Message Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=S;
const string topicName = "codemash.filtering";

Console.WriteLine("Press any key to send messages to the topic...");
Console.ReadKey();

await using ServiceBusClient serviceBusClient = new(connectionString);

await SendMessage(serviceBusClient, topicName, "Hello from the Producer!", "High");
await SendMessage(serviceBusClient, topicName, "Hello from the Producer!", "Low");

2 references | - changes | -authors, -changes
static async Task SendMessage(ServiceBusClient serviceBusClient, string topicName, string messageBody, string priority)
{
    await using ServiceBusSender sender = serviceBusClient.CreateSender(topicName);
    ServiceBusMessage message = new(messageBody);
    message.Subject = priority;
    await sender.SendMessageAsync(message);
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create a Message Producer

Create a Message Consumers

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;Shar
const string topicName = "codemash.filtering";
const string subscriptionName = "LowPriority";

// Create a Service Bus client that will authenticate through the connection string
await using ServiceBusClient serviceBusClient = new(connectionString);

// Create a processor that we can use to process the messages
await using ServiceBusProcessor processor = serviceBusClient.CreateProcessor(topicName, subscriptionName);

// Add handler to process messages
processor.ProcessMessageAsync += async args =>
{
    Console.WriteLine($"Received low-priority message: {args.Message.Body}");
    await args.CompleteMessageAsync(args.Message);
};

// Add handler to process any errors
processor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing
await processor.StartProcessingAsync();
Console.WriteLine("Waiting for low-priority messages, press any key to end processing...");
Console.ReadKey();

// Stop processing
await processor.StopProcessingAsync();
Console.WriteLine("Stopped receiving messages");
```

# Demonstration

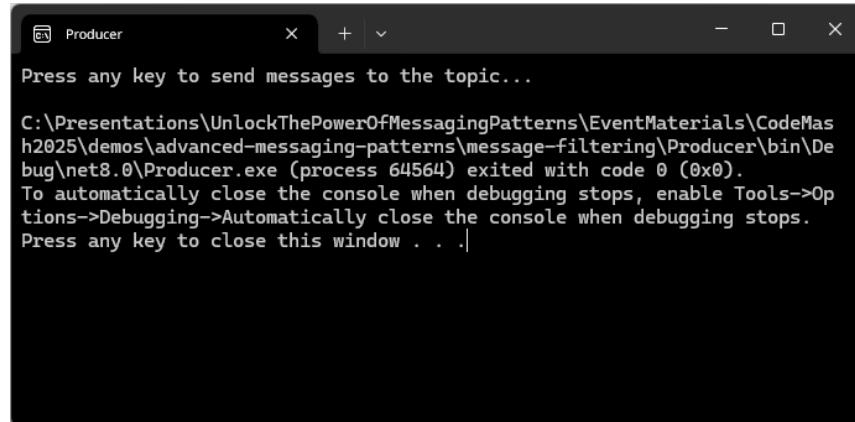
Create Namespace

Create Topic & Subscriptions

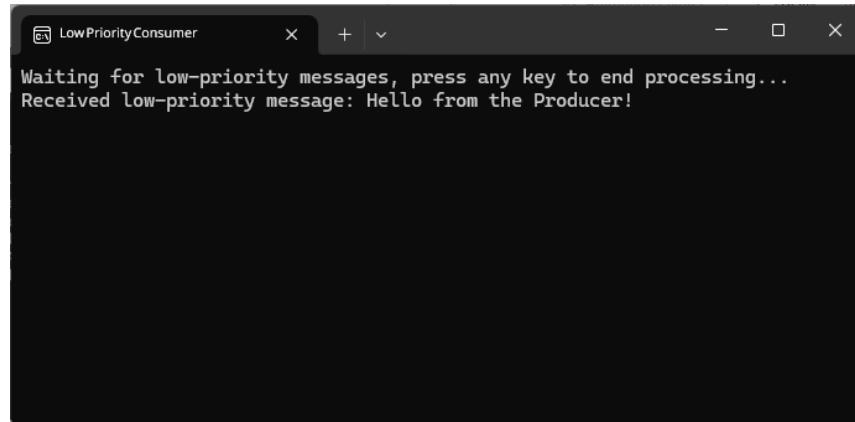
Create a Message Producer

Create Competing Consumers

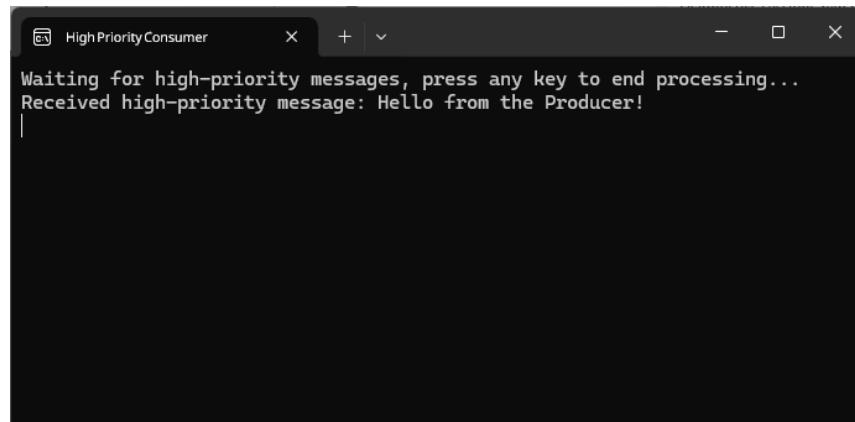
Run the Demo



```
Producer
Press any key to send messages to the topic...
C:\Presentations\UnlockThePowerOfMessagingPatterns\EventMaterials\CodeMash2025\demos\advanced-messaging-patterns\message-filtering\Producer\bin\Debug\net8.0\Producer.exe (process 64564) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .|
```



```
LowPriorityConsumer
Waiting for low-priority messages, press any key to end processing...
Received low-priority message: Hello from the Producer!
```



```
HighPriorityConsumer
Waiting for high-priority messages, press any key to end processing...
Received high-priority message: Hello from the Producer!
```

# Key Points to Remember

- Supports selective message delivery based on specific criteria.
- Enhances efficiency and scalability of message processing.
- Provides flexibility for consumers to adjust their subscription and filters.

# Aggregator Pattern

Routing and Processing

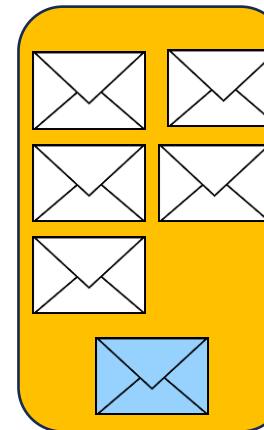
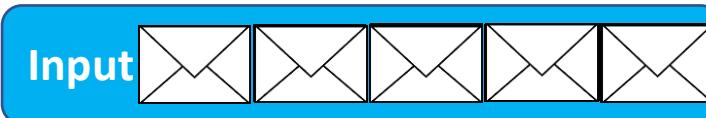
Unlock the Power of Messaging Patterns

# What is the Aggregator Pattern

Multiple message  
collected and processed  
as a single unit

Aggregate related  
information for various  
sources

# Key Components & Flow



Aggregator



Consumer

Producers

# Benefits

Efficiency

Unified View

Scalability

# Drawbacks

Latency

Complexity

Resource  
Utilization

# Use Cases

Batch Processing

Data  
Consolidation

Event Correlation

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create Queues

```
"Queues": [
  {
    "Name": "codemash.aggregator.input",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetterMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "codemash.aggregator.output",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetterMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccess";
const string queueName = "codemash.aggregator.input";

Console.WriteLine("Producer Online...");
Console.WriteLine();

await using ServiceBusClient client = new(connectionString);
await using ServiceBusSender sender = client.CreateSender(queueName);

Console.WriteLine("Press any key to send messages...");
Console.ReadKey();

for (int i = 0; i < 5; i++)
{
    ServiceBusMessage message = new($"Message {i}");
    message.ApplicationProperties["ValueToAggregate"] = i;
    await sender.SendMessageAsync(message);
    Console.WriteLine($"Sent message {i}");
}

Console.WriteLine("Done!");
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string inputQueueName = "codemash.aggregator.input";
const string outputQueueName = "codemash.aggregator.output";

Console.WriteLine("Aggregator Online...");
Console.WriteLine();

List<ServiceBusReceivedMessage> receivedMessages = await ReceivedMessagesAsync();
int aggregatedValue = AggregateMessages(receivedMessages);
await SendOutputMessageAsync(aggregatedValue);

Console.WriteLine("Done!");
Console.ReadKey();
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

```
using Azure.Messaging.ServiceBus;

async Task<List<ServiceBusReceivedMessage>> ReceivedMessagesAsync()
{
    await using ServiceBusClient serviceBusClient = new(connectionString);
    await using ServiceBusReceiver serviceBusReceiver = serviceBusClient.CreateReceiver(inputQueueName);
    List<ServiceBusReceivedMessage> receivedMessages = [];
    Console.WriteLine("Receiving messages...");
    while (receivedMessages.Count < 5)
    {
        ServiceBusReceivedMessage receivedMessage = await serviceBusReceiver.ReceiveMessageAsync();
        receivedMessages.Add(receivedMessage);
        Console.WriteLine("\tReceived message: " + receivedMessage.Body);
        await serviceBusReceiver.CompleteMessageAsync(receivedMessage);
    }
    return receivedMessages;
}

Console.ReadKey();
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string inputQueueName = "codemash.aggregator.input";
const string outputQueueName = "codemash.aggregator.output";

Console.WriteLine("Aggregator Online...");
int AggregateMessages(List<ServiceBusReceivedMessage> messages)
{
    Console.WriteLine("Aggregating messages...");
    return messages.ToArray().Select(x => (int)x.ApplicationProperties["ValueToAggregate"]).Sum();
}

await ServiceBusProcessor.StartAggregation();

Console.WriteLine("Done!");
Console.ReadKey();
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string inputQueueName = "codemash.aggregator.input";
const string outputQueueName = "codemash.aggregator.output";

async Task SendOutputMessageAsync(int aggregatedValue)
{
    Console.WriteLine("Sending aggregated value...");
    await using ServiceBusClient serviceBusClient = new(connectionString);
    await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(outputQueueName);
    Console.WriteLine("Sending aggregated value...");
    ServiceBusMessage outputMessage = new($"Aggregated value: {aggregatedValue}");
    await serviceBusSender.SendMessageAsync(outputMessage);
}

Console.WriteLine("Done!");
Console.ReadKey();
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

Create the Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "codemash.aggregator.output";

Console.WriteLine("Consumer Online...");
Console.WriteLine();

await using ServiceBusClient client = new(connectionString);
await using ServiceBusProcessor processor = client.CreateProcessor(queueName);

processor.ProcessMessageAsync += args =>
{
    Console.WriteLine();
    Console.WriteLine("Received message: " + args.Message.Body);
    return args.CompleteMessageAsync(args.Message);
};

processor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception);
    return Task.CompletedTask;
};

await processor.StartProcessingAsync();
Console.WriteLine("Ready to receive aggregation outputs...");
Console.WriteLine("Press any key to stop...");
Console.ReadKey();
await processor.StopProcessingAsync();
```

# Demonstration

Create Namespace

Create Queues

Create the Producer

Create the Aggregator

Create the Consumer

Run the Demo

```
Producer Online...
Press any key to send messages...
Sent message 0
Sent message 1
Sent message 2
Sent message 3
Sent message 4
Done!
```

```
Aggregator Online...
Receiving messages...
Received message: Message 0
Received message: Message 1
Received message: Message 2
Received message: Message 3
Received message: Message 4
Aggregating messages...
Sending aggregated value...
Sending aggregated value...
Done!
```

```
Consumer Online...
Ready to receive aggregation outputs...
Press any key to stop...
Received message: Aggregated value: 10
```

# Key Points to Remember

- Aggregates related messages for batch processing and data consolidation.
- Enhances efficiency and scalability by reducing the overhead of processing individual messages.
- Provides a unified view of related information, simplifying analysis and decision-making.

# Scatter-Gather Pattern

Routing and Processing

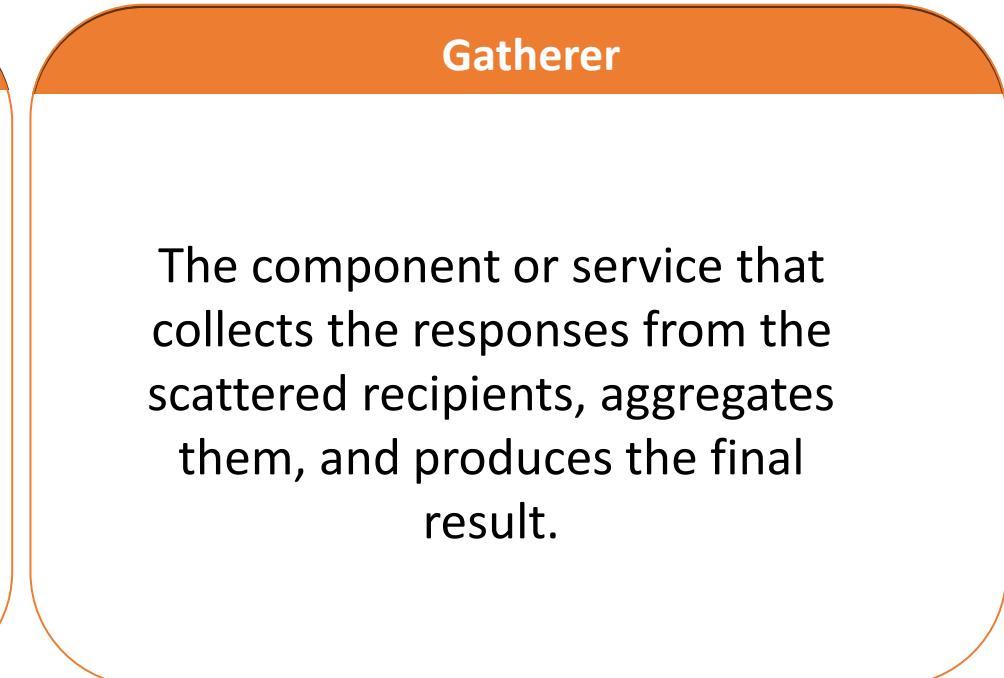
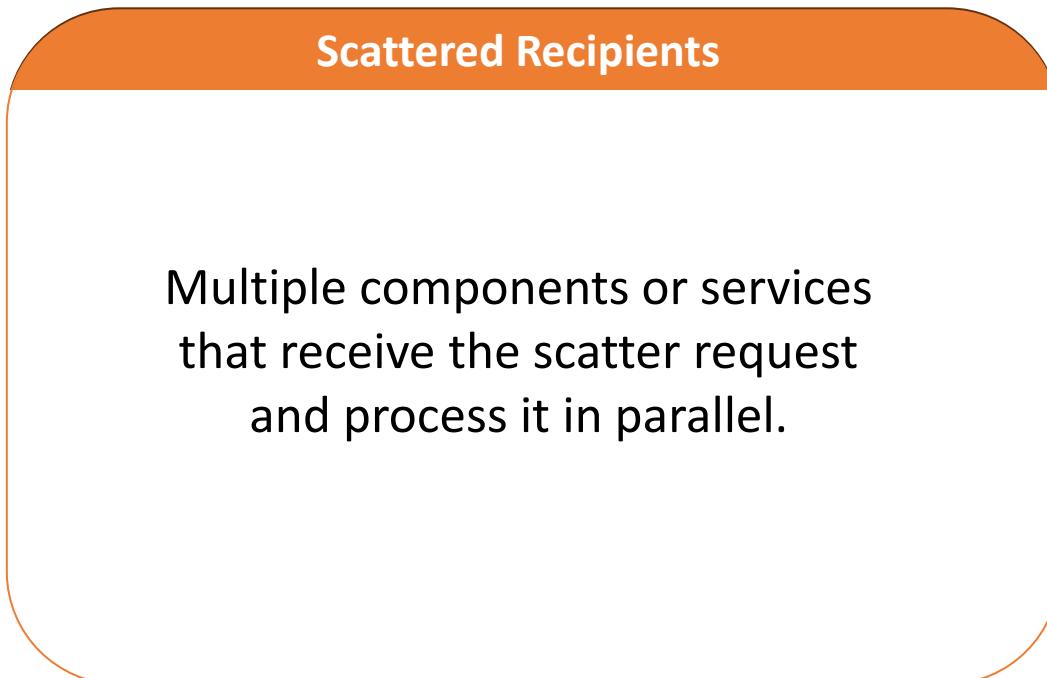
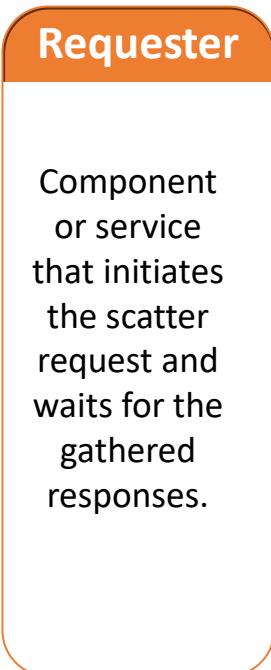
Unlock the Power of Messaging Patterns

# What is the Aggregator Pattern

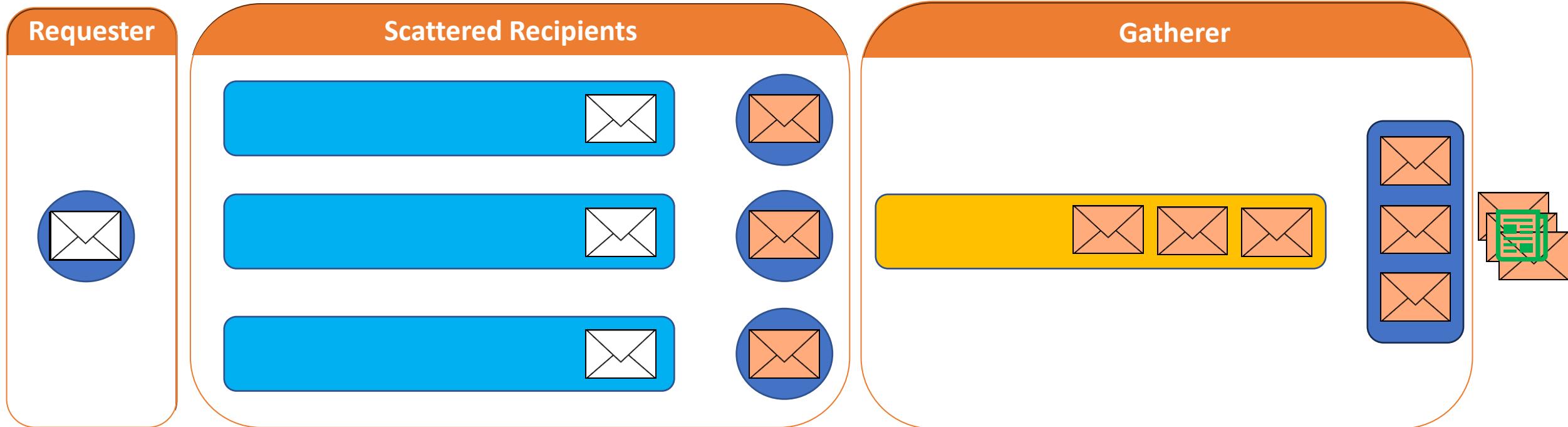
Request sent to multiple recipients and responses are aggregated

Parallel processing and aggregation of results

# Key Components



# Flow



# Benefits

Efficiency

Scalability

Resilience

# Drawbacks

Complexity

Latency

Scalability

# Use Cases

Parallel  
Processing

Data Aggregation

Load Balancing

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

```
"Topics": [
  {
    "Name": "scatter-gather.request",
    "Properties": {
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "RequiresDuplicateDetection": false
    },
    "Subscriptions": [
      {
        "Name": "recipient-1",
        "Properties": {
          "DeadLetteringOnMessageExpiration": false,
          "DefaultMessageTimeToLive": "PT1H",
          "LockDuration": "PT1M",
          "MaxDeliveryCount": 3,
          "ForwardDeadletteredMessagesTo": "",
          "ForwardTo": "",
          "RequiresSession": false
        }
      },
      {
        "Name": "recipient-2",
        "Properties": {
          "DeadLetteringOnMessageExpiration": false,
          "DefaultMessageTimeToLive": "PT1H",
          "LockDuration": "PT1M",
          "MaxDeliveryCount": 3,
          "ForwardDeadletteredMessagesTo": "",
          "ForwardTo": "",
          "RequiresSession": false
        }
      },
      {
        "Name": "recipient-3",
        "Properties": {
          "DeadLetteringOnMessageExpiration": false,
          "DefaultMessageTimeToLive": "PT1H",
          "LockDuration": "PT1M",
          "MaxDeliveryCount": 3,
          "ForwardDeadletteredMessagesTo": "",
          "ForwardTo": "",
          "RequiresSession": false
        }
      }
    ]
  }
]
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

```
"Queues": [
  {
    "Name": "scatter-gather.gather",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetterMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "scatter-gather.response",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetterMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
],
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

```
static async Task SendRequestAsync()
{
    Console.WriteLine("Press any key to start the scatter-gather process...");
    Console.ReadKey(true);

    // Create a Service Bus client and sender
    await using ServiceBusClient serviceBusClient = new(connectionString);
    await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(requestTopicName);

    // Send a scatter request message
    ServiceBusMessage message = new("Scatter Request");
    await serviceBusSender.SendMessageAsync(message);

    Console.WriteLine("Scatter Request sent.");
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

```
using Azure.Messaging.ServiceBus;  
  
const string connectionString = "Endpoint=sb://127.0.0.1;Sha  
const string requestTopicName = "scatter-gather.request";  
const string responseQueueName = "scatter-gather.response";  
  
await SendRequestAsync();  
await ReceiveResponse();
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
public static class ScatteredRequestHelper
{
    3 references | 0 changes | 0 authors, 0 changes
    public static async Task ProcessScatteredRequests(
        string recipientName,
        string serviceBusConnectionString,
        string scatterTopicName,
        string recipientSubscriptionName,
        string gathererQueueName)
    {
        Random random = new();

        // Create the ServiceBusClient, ServiceBusProcessor, and ServiceBusSender objects
        await using ServiceBusClient serviceBusClient = new(serviceBusConnectionString);
        await using ServiceBusProcessor requestProcessor = serviceBusClient.CreateProcessor(scatterTopicName, recipientSubscriptionName);
        await using ServiceBusSender responseSender = serviceBusClient.CreateSender(gathererQueueName);

        // Add an event handler to process messages
        requestProcessor.ProcessMessageAsync += async args =>
        {
            string body = args.Message.Body.ToString();
            Console.WriteLine($"Received message: {body}");

            // Send a response message to the gatherer queue
            ServiceBusMessage responseMessage = new("Scatter Recipient Response");
            responseMessage.ApplicationProperties["RecipientName"] = recipientName;
            responseMessage.ApplicationProperties["Response"] = random.Next(1, 100);
            await responseSender.SendMessageAsync(responseMessage);
            Console.WriteLine($"Sent recipient response message: {responseMessage.Body}");

            await args.CompleteMessageAsync(args.Message);
        };

        // Add an event handler to process any errors
        requestProcessor.ProcessErrorAsync += args =>
        {
            Console.WriteLine(args.Exception.ToString());
            return Task.CompletedTask;
        };

        // Start processing
        await requestProcessor.StartProcessingAsync();
        Console.WriteLine();
        Console.WriteLine($"{recipientName} is listening for messages; press any key to stop...");
        Console.ReadKey(true);

        // Stop processing
        await requestProcessor.StopProcessingAsync();
    }
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
public static class ScatteredRequestHelper
{
    // references | 0 changes | 0 authors, 0 changes
    public static async Task ProcessScatteredRequests(
        string recipientName,
        string serviceBusConnectionString,
        string scatterTopicName,
        string recipientSubscriptionName,
        string gathererQueueName)
    {
        Random random = new();

        // Create the ServiceBusClient, ServiceBusProcessor, and ServiceBusSender objects
        await using ServiceBusClient serviceBusClient = new(serviceBusConnectionString);
        await using ServiceBusProcessor requestProcessor = serviceBusClient.CreateProcessor(scatterTopicName, recipientSubscriptionName);
        await using ServiceBusSender responseSender = serviceBusClient.CreateSender(gathererQueueName);

        requestProcessor.ProcessMessageAsync += async args =>
        {
            string body = args.Message.Body.ToString();
            Console.WriteLine($"Received message: {body}");

            // Send a response message to the gatherer queue
            ServiceBusMessage responseMessage = new("Scatter Recipient Response");
            responseMessage.ApplicationProperties["RecipientName"] = recipientName;
            responseMessage.ApplicationProperties["Response"] = random.Next(1, 100);
            await responseSender.SendMessageAsync(responseMessage);
            Console.WriteLine($"Sent recipient response message: {responseMessage.Body}");

            await args.CompleteMessageAsync(args.Message);
        };

        // Add an event handler to process any errors
        requestProcessor.ProcessErrorAsync += args =>
        {
            Console.WriteLine(args.Exception.ToString());
            return Task.CompletedTask;
        };

        // Start processing
        await requestProcessor.StartProcessingAsync();
        Console.WriteLine();
        Console.WriteLine($"{recipientName} is listening for messages; press any key to stop...");
        Console.ReadKey(true);

        // Stop processing
        await requestProcessor.StopProcessingAsync();
    }
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
public static class ScatteredRequestHelper
{
    3 references | 0 changes | 0 authors, 0 changes
    public static async Task ProcessScatteredRequests(
        string recipientName,
        string serviceBusConnectionString,
        string scatterTopicName,
        string recipientsSubscriptionName,
        string gathererQueueName)
    {
        // Add an event handler to process messages
        requestProcessor.ProcessMessageAsync += async args =>
        {
            string body = args.Message.Body.ToString();
            Console.WriteLine($"Received message: {body}");

            // Send a response message to the gatherer queue
            ServiceBusMessage responseMessage = new("Scatter Recipient Response");
            responseMessage.ApplicationProperties["RecipientName"] = recipientName;
            responseMessage.ApplicationProperties["Response"] = random.Next(1, 100);
            await responseSender.SendMessageAsync(responseMessage);
            Console.WriteLine($"Sent recipient response message: {responseMessage.Body}");

            await args.CompleteMessageAsync(args.Message);
        };
        Console.ReadKey(true);

        // Stop processing
        await requestProcessor.StopProcessingAsync();
    }
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
public static class ScatteredRequestHelper
{
    3 references | 0 changes | 0 authors, 0 changes
    public static async Task ProcessScatteredRequests(
        string recipientName,
        string serviceBusConnectionString,
        string scatterTopicName,
        string recipientsSubscriptionName,
        string gathererQueueName)
    {
        Random random = new();

        // Create the ServiceBusClient, ServiceBusProcessor, and ServiceBusSender objects
        await using ServiceBusClient serviceBusClient = new(serviceBusConnectionString);
        await using ServiceBusProcessor requestProcessor = serviceBusClient.CreateProcessor(scatterTopicName, recipientSubscriptionName);
        await using ServiceBusSender responseSender = serviceBusClient.CreateSender(gathererQueueName);

        // Add an event handler to process any errors
        requestProcessor.ProcessErrorAsync += args =>
        {
            Console.WriteLine(args.Exception.ToString());
            return Task.CompletedTask;
        };

        // Start processing
        await requestProcessor.StartProcessingAsync();
        Console.WriteLine();
        Console.WriteLine($"{recipientName} is listening for messages; press any key to stop...");
        Console.ReadKey(true);

        // Stop processing
        await requestProcessor.StopProcessingAsync();
    }
}
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
using ScatteredRecipientHelper;

const string recipientName = "Scattered Recipient 1";
const string connectionString = "Endpoint=sb://127.0.0.1;Sha
const string topicName = "scatter-gather.request";
const string subscriptionName = "recipient-1";
const string gathererQueueName = "scatter-gather.gather";

await ScatteredRequestHelper.ProcessScatteredRequests(
    recipientName,
    connectionString,
    topicName,
    subscriptionName,
    gathererQueueName);
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
using ScatteredRecipientHelper;

const string recipientName = "Scattered Recipient 1";
const using ScatteredRecipientHelper;
const
const
const
const
const
await
reci
conn
topi
subs
gath
const string recipientName = "Scattered Recipient 2";
const string connectionString = "Endpoint=sb://127.0.0.1:5671";
const string topicName = "scatter-gather.request";
const string subscriptionName = "recipient-2";
const string gathererQueueName = "scatter-gather.gather";

await ScatteredRequestHelper.ProcessScatteredRequests(
    recipientName,
    connectionString,
    topicName,
    subscriptionName,
    gathererQueueName);
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

Create Queues

Create Requester

Create Recipients

```
using ScatteredRecipientHelper;

const string recipientName = "Scattered Recipient 1";
const string connectionString = "Endpoint=sb://127.0.0.1;Sha
const string topicName = "scatter-gather.request";
const string subscriptionName = "recipient-1";
const string recipientName = "Scattered Recipient 2";
const string connectionString = "Endpoint=sb://127.0.0.1;Sha
const string topicName = "scatter-gather.request";
const string subscriptionName = "recipient-2";
const string recipientName = "Scattered Recipient 3";
const string connectionString = "Endpoint=sb://127.0.0.1;Sha
const string topicName = "scatter-gather.request";
const string subscriptionName = "recipient-3";
const string gathererQueueName = "scatter-gather.gather";

await ScatteredRequestHelper.ProcessScatteredRequests(
    recipientName,
    connectionString,
    topicName,
    subscriptionName,
    gathererQueueName);
```

# Demonstration

Create Namespace

Create Topic & Subscriptions

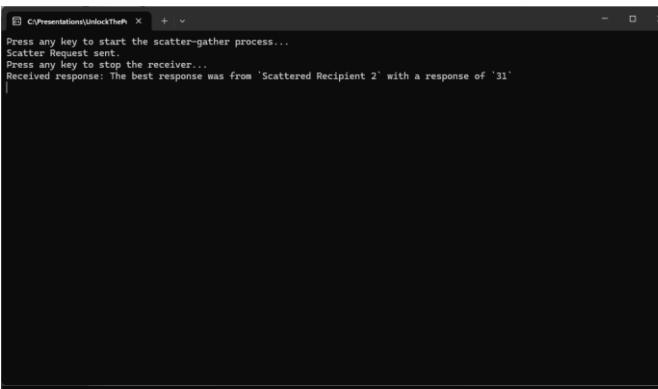
Create Queues

Create Requester

Create Recipients

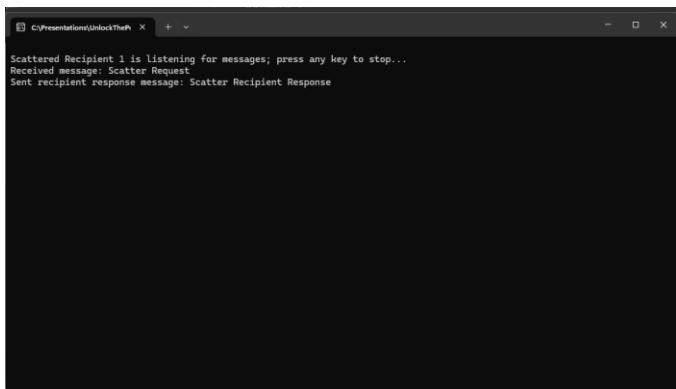
Run the Demo

# Requester

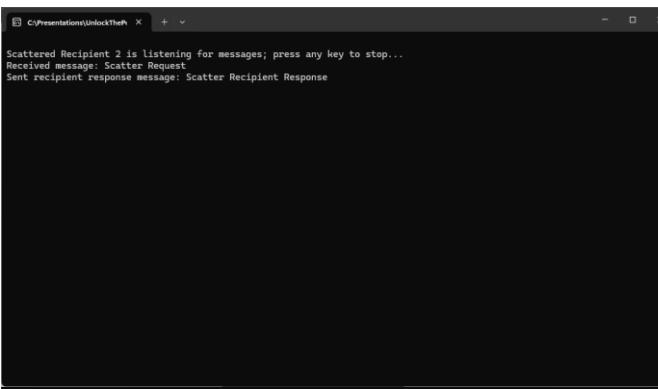


```
Press any key to start the scatter-gather process...
Scatter Request sent.
Press any key to stop the receiver...
Received response: The best response was from 'Scattered Recipient 2' with a response of '31'
```

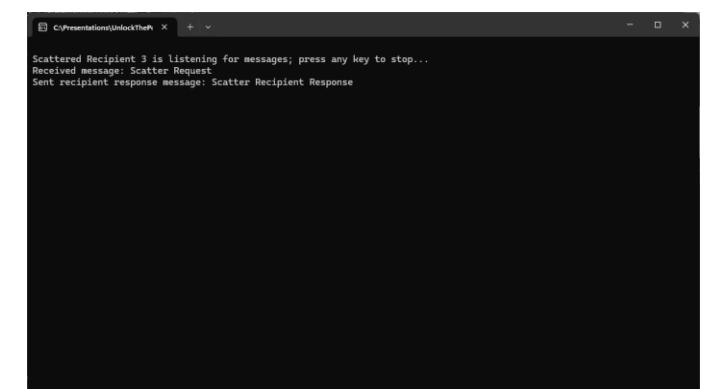
# Recipients



```
Scattered Recipient 1 is listening for messages; press any key to stop...
Received message: Scatter Request
Sent recipient response message: Scatter Recipient Response
```

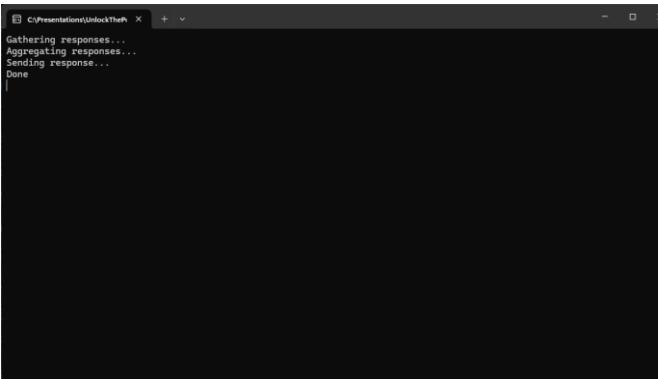


```
Scattered Recipient 2 is listening for messages; press any key to stop...
Received message: Scatter Request
Sent recipient response message: Scatter Recipient Response
```



```
Scattered Recipient 3 is listening for messages; press any key to stop...
Received message: Scatter Request
Sent recipient response message: Scatter Recipient Response
```

# Gatherer



```
Gathering responses...
Aggregating responses...
Sending response...
Done
```

Unlock the Power of Messaging Patterns

# Key Points to Remember

- Aggregates related messages for batch processing and data consolidation.
- Enhances efficiency and scalability by reducing the overhead of processing individual messages.
- Provides a unified view of related information, simplifying analysis and decision-making.

# Wrap-Up

Routing and Processing

Unlock the Power of Messaging Patterns

# Routing and Processing Wrap-Up

Message Routing

Dead Letter  
Queues

Message Filtering

Aggregator

Scatter-Gather

# Preview: Advanced Processing Techniques

Idempotent Receiver

Message Scheduling

Transactional Queues

Claim Check

# Advanced Processing Techniques

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Idempotent Receiver Pattern

Advanced Processing Techniques

Unlock the Power of Messaging Patterns

# What is the Idempotent Receiver Pattern

Ensures message is processed exactly once

Prevent duplicate processing of messages

# Key Components and Flow

Unique Message  
Identifiers

State Storage

Idempotent  
Processing Logic

# Key Components and Flow

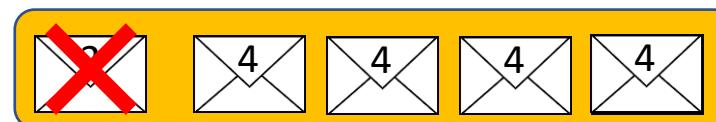
Unique Message  
Identifiers



State Storage



Idempotent  
Processing Logic



# Benefits

Data Integrity

Reliability

Efficiency

# Drawbacks

Complexity

State  
Management

# Use Cases

Payment  
Processing

Order  
Management

Inventory  
Updates

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create Queue

```
"Queues": [
  {
    "Name": "idempotent-receiver",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

# Demonstration

Create Namespace

Create Queue

Create Entity

```
public class MyMessage
{
    5 references | 0 changes | 0 authors, 0 changes
    public Guid MessageId { get; set; } = Guid.NewGuid();
    3 references | 0 changes | 0 authors, 0 changes
    public required string Content { get; set; }
}
```

# Demonstration

Create Namespace

Create Queue

Create Entity

Create Producer

```
using Azure.Messaging.ServiceBus;
using Entities;
using System.Text.Json;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "idempotent-receiver";

// Prompt the user to start sending messages
Console.WriteLine("Press any key to start sending messages...");
Console.ReadKey(true);

await using ServiceBusClient serviceBusClient = new(connectionString);
await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(queueName);

// Send messages to the queues
Console.WriteLine();
Console.WriteLine("Sending messages to the queues...");

MyMessage message1 = new() { Content = "First attempt" };
MyMessage message2 = new() { MessageId = message1.MessageId, Content = "Duplicate attempt" };

await serviceBusSender.SendMessageAsync(new ServiceBusMessage(JsonSerializer.Serialize(message1)));
await serviceBusSender.SendMessageAsync(new ServiceBusMessage(JsonSerializer.Serialize(message2)));

// Wait for the user to press a key before exiting
Console.WriteLine();
Console.WriteLine("Finished sending messages. Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

Create Namespace

Create Queue

Create Entity

Create Producer

Create Receiver

```
using Azure.Messaging.ServiceBus;
using Entities;
using Microsoft.Extensions.Caching.Memory;
using System.Text.Json;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "idempotent-receiver";

MemoryCache _cache = new(new MemoryCacheOptions());

// Create a Service Bus client and processor to process messages
await using ServiceBusClient serviceBusClient = new(connectionString);
await using ServiceBusProcessor processor = serviceBusClient.CreateProcessor(queueName);

// Add an event handler to the processor to process messages
processor.ProcessMessageAsync += args =>
{
    MyMessage? message = JsonSerializer.Deserialize<MyMessage>(args.Message.Body.ToString());

    if (message is not null)
    {
        if (!_cache.TryGetValue(message.MessageId, out _))
        {
            // Process the message
            Console.WriteLine($"Processing message: {message.Content}");

            // Mark the message as processed
            _cache.Set(message.MessageId, true, TimeSpan.FromMinutes(10));
        }
        else
        {
            Console.WriteLine($"Message with ID {message.MessageId} has already been processed.");
        }

        // Complete the message
        await args.CompleteMessageAsync(args.Message);
    }
    else
    {
        await args.DeadLetterMessageAsync(args.Message);
    }
};

// Add an error handler to the processor to handle any errors when receiving messages
processor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing messages from the queue
await processor.StartProcessingAsync();
Console.WriteLine();
Console.WriteLine("Press any key to stop receiving messages...");
Console.ReadKey(true);

// Stop processing messages from the queue
await processor.StopProcessingAsync();
```

# Demonstration

Create Namespace

Create Queue

Create Entity

Create Producer

Create Receiver

```
using Azure.Messaging.ServiceBus;
using Entities;
using Microsoft.Extensions.Caching.Memory;
using System.Text.Json;

MemoryCache _cache = new(new MemoryCacheOptions());

await using ServiceBusProcessor processor = servicebusClient.CreateProcessor(queueName);

// Add an event handler to the processor to process messages
processor.ProcessMessageAsync += async args =>
{
    MyMessage? message = JsonSerializer.Deserialize<MyMessage>(args.Message.Body.ToString());

    if (message is not null)
    {
        if (!_cache.TryGetValue(message.MessageId, out _))
        {
            // Process the message
            Console.WriteLine($"Processing message: {message.Content}");

            // Mark the message as processed
            _cache.Set(message.MessageId, true, TimeSpan.FromMinutes(10));
        }
        else
        {
            Console.WriteLine($"Message with ID {message.MessageId} has already been processed.");
        }

        // Complete the message
        await args.CompleteMessageAsync(args.Message);
    }
    else
    {
        await args.DeadLetterMessageAsync(args.Message);
    }
};

// Add an error handler to the processor to handle any errors when receiving messages
processor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing messages from the queue
await processor.StartProcessingAsync();
Console.WriteLine();
Console.WriteLine("Press any key to stop receiving messages...");
Console.ReadKey(true);

// Stop processing messages from the queue
await processor.StopProcessingAsync();
```

# Demonstration

Create Namespace

Create Queue

Create Entity

Create Producer

Create Receiver

```
// Add an event handler to the processor to process messages
processor.ProcessMessageAsync += async args =>
{

    MyMessage? message = JsonSerializer.Deserialize<MyMessage>(args.Message.Body.ToString());

    if (message is not null)
    {
        if (!_cache.TryGetValue(message.MessageId, out _))
        {
            // Process the message
            Console.WriteLine($"Processing message: {message.Content}");

            // Mark the message as processed
            _cache.Set(message.MessageId, true, TimeSpan.FromMinutes(10));
        }
        else
        {
            Console.WriteLine($"Message with ID {message.MessageId} has already been processed.");
        }

        // Complete the message
        await args.CompleteMessageAsync(args.Message);

    }
    else
    {
        await args.DeadLetterMessageAsync(args.Message);
    }
};
```

# Demonstration

Create Namespace

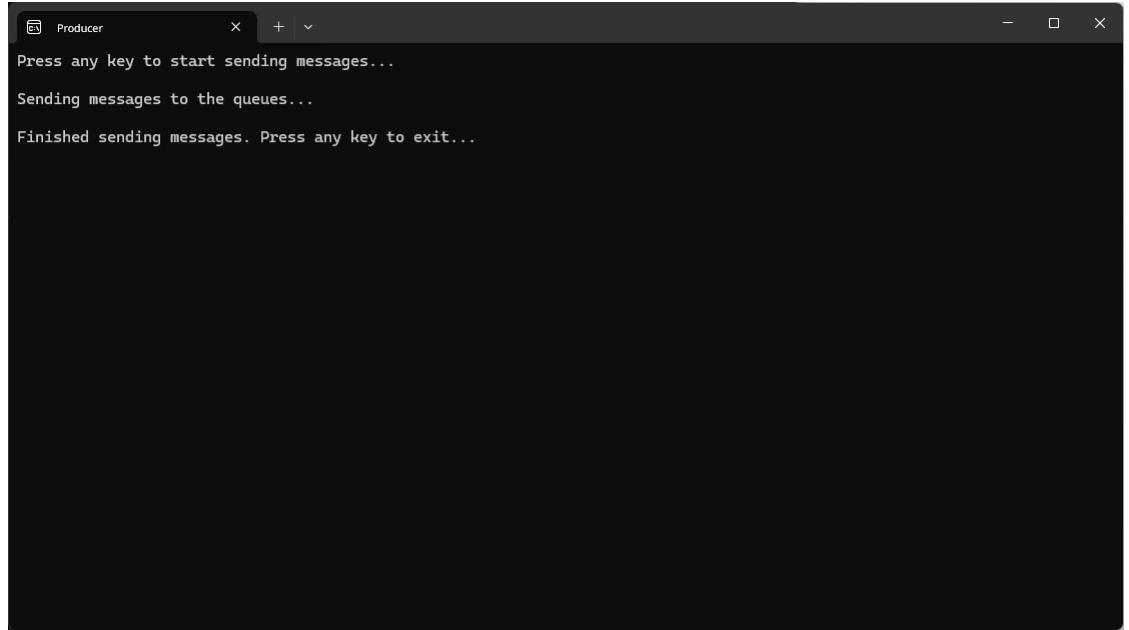
Create Queue

Create Entity

Create Producer

Create Receiver

Run the Demo

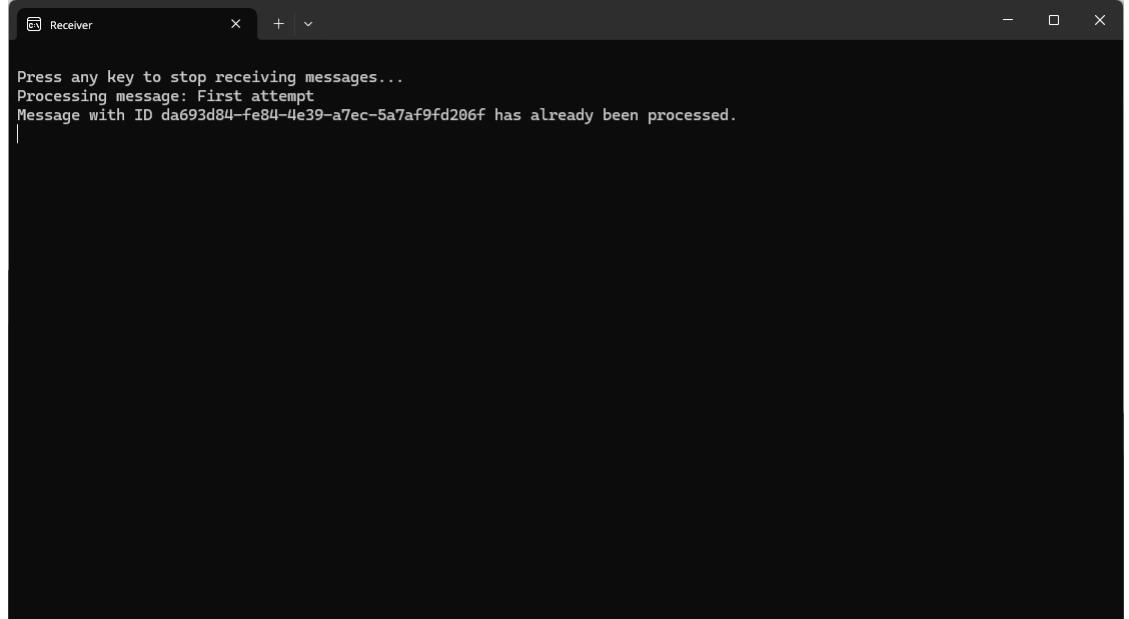


Producer

Press any key to start sending messages...

Sending messages to the queues...

Finished sending messages. Press any key to exit...



Receiver

Press any key to stop receiving messages...

Processing message: First attempt

Message with ID da693d84-fe84-4e39-a7ec-5a7af9fd206f has already been processed.

# Key Points to Remember

Use Unique  
Identifiers

Maintain  
Processing State

Implement  
Idempotent Logic

Monitor and  
Audit

# Message Scheduling Pattern

Advanced Processing Techniques

Unlock the Power of Messaging Patterns

# What is the Message Scheduling Pattern

Delaying the processing  
of a message.

Enable system to handle  
tasks at appropriate  
times

# Key Components and Flow

Scheduler

Queue

Time-based  
Triggers

# Key Components and Flow

Scheduler

Queue

Time-based  
Triggers



# Benefits

Controlled  
Processing

Resource  
Optimization

Improved  
Reliability

# Drawbacks

Complexity

Potential Delays

# Use Cases

Delayed  
Notifications

Batch Processing

Retry Logic

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create Namespace

Create Queue

```
"Queues": [  
  {  
    "Name": "message-scheduling",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLetteredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": false  
    }  
  ]
```

# Demonstration

Create Namespace

Create Queue

Create Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "message-scheduling";

// Prompt the user to start sending messages
Console.WriteLine("Press any key to start sending messages.");
Console.ReadKey(true);

await using ServiceBusClient serviceBusClient = new(connectionString);
ServiceBusSender sender = serviceBusClient.CreateSender(queueName);

// Send a message that will be processed at a later time
ServiceBusMessage message = new($"Hello, Scheduling! Current Time: {DateTime.UtcNow}")
{
    ScheduledEnqueueTime = DateTimeOffset.UtcNow.AddSeconds(5)
};
await sender.SendMessageAsync(message);
Console.WriteLine($"Message '{message.Body}' scheduled to be sent at: {message.ScheduledEnqueueTime}");

// Send a message that will be processed right away
message = new ServiceBusMessage($"Hello, World! Current Time: {DateTime.UtcNow}");
await sender.SendMessageAsync(message);
Console.WriteLine($"Message '{message.Body}' sent immediately.");

Console.WriteLine();
Console.WriteLine("Finished sending messages. Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

Create Namespace

Create Queue

Create Producer

Create Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccess";
const string queueName = "message-scheduling";

await using ServiceBusClient serviceBusClient = new(connectionString);
await using ServiceBusProcessor serviceBusProcessor = serviceBusClient.CreateProcessor(queueName);

serviceBusProcessor.ProcessMessageAsync += async processMessageEventArgs =>
{
    string body = processMessageEventArgs.Message.Body.ToString();
    Console.WriteLine($"Processing message: {body}");
    Console.WriteLine(processMessageEventArgs.Message.EnqueueTime);
    await processMessageEventArgs.CompleteMessageAsync(processMessageEventArgs.Message);
};

serviceBusProcessor.ProcessErrorAsync += processErrorEventArgs =>
{
    Console.WriteLine($"Error when receiving messages: {processErrorEventArgs.Exception}");
    return Task.CompletedTask;
};

await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Press any key to stop receiving messages...");
Console.ReadKey(true);

await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine("Stopped receiving messages. Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

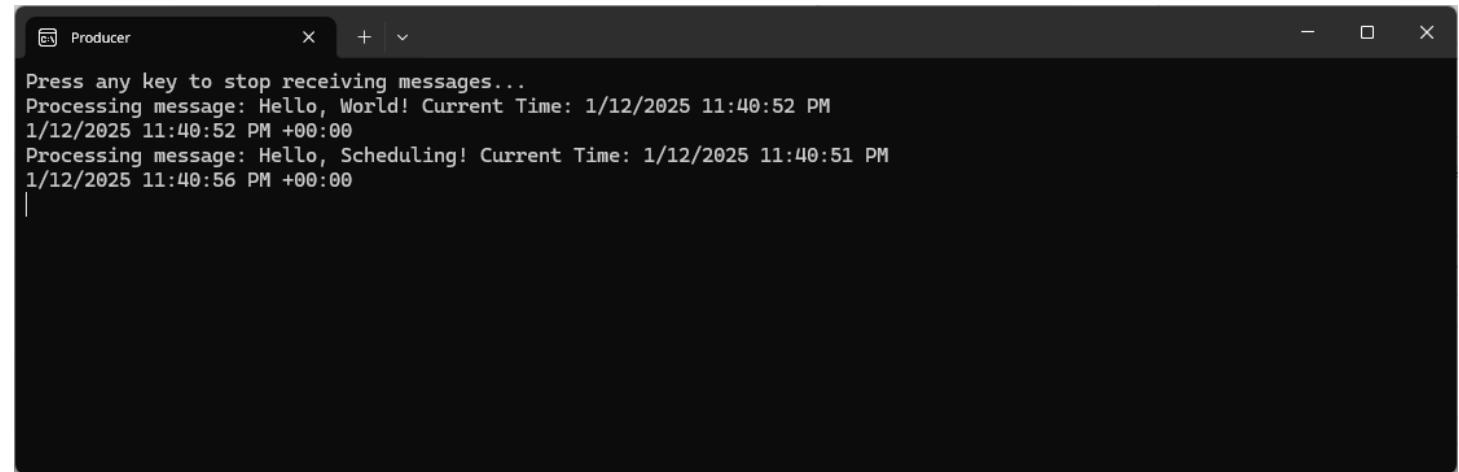
Create Namespace

Create Queue

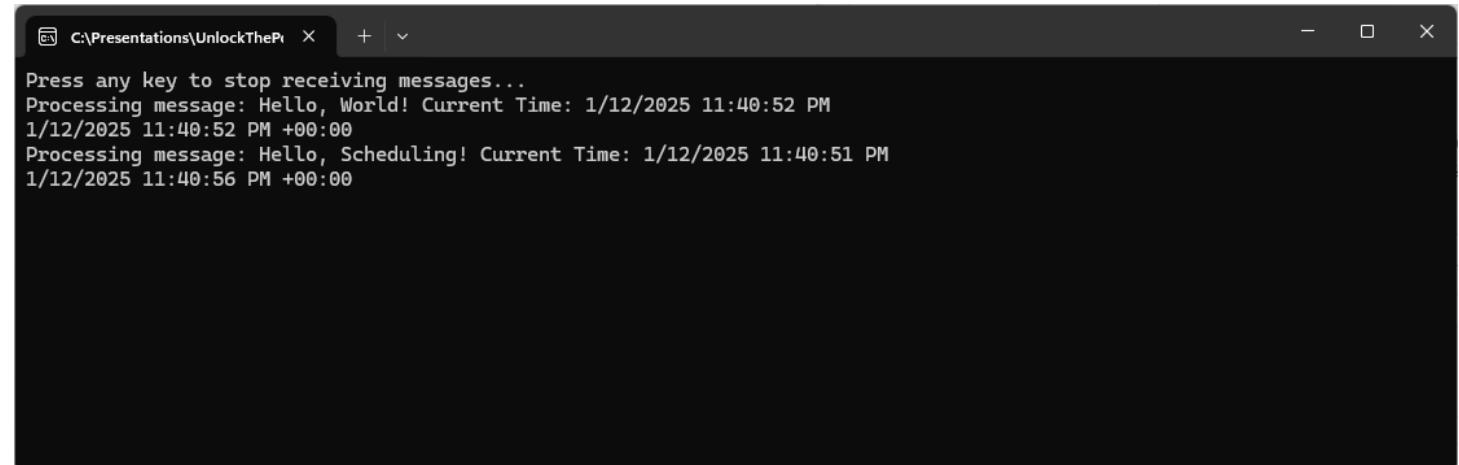
Create Producer

Create Consumer

Run the Demo



```
Producer
Press any key to stop receiving messages...
Processing message: Hello, World! Current Time: 1/12/2025 11:40:52 PM
1/12/2025 11:40:52 PM +00:00
Processing message: Hello, Scheduling! Current Time: 1/12/2025 11:40:51 PM
1/12/2025 11:40:56 PM +00:00
```



```
C:\Presentations\UnlockTheP...
Press any key to stop receiving messages...
Processing message: Hello, World! Current Time: 1/12/2025 11:40:52 PM
1/12/2025 11:40:52 PM +00:00
Processing message: Hello, Scheduling! Current Time: 1/12/2025 11:40:51 PM
1/12/2025 11:40:56 PM +00:00
```

# Key Points to Remember

Define Scheduling Criteria

Implement Scheduling Logic

Monitor and Manage

Handle Failures

# Transactional Queues Pattern

Advanced Processing Techniques

Unlock the Power of Messaging Patterns

# What is the Transactional Queues Pattern

Ensures messages processed within a transaction

Guarantees reliable and consistent message processing

# Benefits

Reliability

Consistency

Isolation

Durability

# Drawbacks

Complexity

Performance  
Overhead

Resource  
Management

# Use Cases

Financial  
Transactions

Order Processing

Distributed  
Systems

# Demonstration

Create Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create the Namespace

Create the Queues

```
{  
  "Queues": [  
    {  
      "Name": "transactional-queues.initiator",  
      "Properties": {  
        "DeadLetteringOnMessageExpiration": false,  
        "DefaultMessageTimeToLive": "PT1H",  
        "DuplicateDetectionHistoryTimeWindow": "PT20S",  
        "ForwardDeadletteredMessagesTo": "",  
        "ForwardTo": "",  
        "LockDuration": "PT1M",  
        "MaxDeliveryCount": 3,  
        "RequiresDuplicateDetection": false,  
        "RequiresSession": false  
      }  
    },  
    {  
      "Name": "transactional-queues.participant-1",  
      "Properties": {  
        "DeadLetteringOnMessageExpiration": false,  
        "DefaultMessageTimeToLive": "PT1H",  
        "DuplicateDetectionHistoryTimeWindow": "PT20S",  
        "ForwardDeadletteredMessagesTo": "",  
        "ForwardTo": "",  
        "LockDuration": "PT1M",  
        "MaxDeliveryCount": 3,  
        "RequiresDuplicateDetection": false,  
        "RequiresSession": false  
      }  
    },  
    {  
      "Name": "transactional-queues.participant-2",  
      "Properties": {  
        "DeadLetteringOnMessageExpiration": false,  
        "DefaultMessageTimeToLive": "PT1H",  
        "DuplicateDetectionHistoryTimeWindow": "PT20S",  
        "ForwardDeadletteredMessagesTo": "",  
        "ForwardTo": "",  
        "LockDuration": "PT1M",  
        "MaxDeliveryCount": 3,  
        "RequiresDuplicateDetection": false,  
        "RequiresSession": false  
      }  
    }  
  ]  
}
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";

// Prompt the user to start sending messages
Console.WriteLine("Press any key to start sending messages...");
Console.ReadKey(true);

// Create a Service Bus client and sender
await using ServiceBusClient serviceBusClient = new(connectionString);
await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(initiatorQueueName);

// Send a message to the initiator queue
await serviceBusSender.SendMessageAsync(new ServiceBusMessage("Initiate Transaction"));
Console.WriteLine("Message sent to the initiator queue.");

// Prompt the user to end the application
Console.WriteLine("Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

```
using Azure.Messaging.ServiceBus;
using System.Transactions;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";
const string firstTransactionParticipantQueueName = "transactional-queues.participant-1";
const string secondTransactionParticipantQueueName = "transactional-queues.participant-2";

// Create a Service Bus client
ServiceBusClientOptions serviceBusClientOptions = new() { EnableCrossEntityTransactions = true };
await using ServiceBusClient client = new(connectionString, serviceBusClientOptions);

// Create a Service Bus receiver and senders
ServiceBusReceiver initiatorReceiver = client.CreateReceiver(initiatorQueueName);
ServiceBusSender participant1Sender = client.CreateSender(firstTransactionParticipantQueueName);
ServiceBusSender participant2Sender = client.CreateSender(secondTransactionParticipantQueueName);

// Receive a message from the initiator queue
Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

// Create a TransactionScope object
using TransactionScope transactionScope = new(TransactionScopeAsyncFlowOption.Enabled);

// Complete the initiator message
await initiatorReceiver.CompleteMessageAsync(receivedMessage);

// Send a message to the first participant queue
await participant1Sender.SendMessageAsync(new ServiceBusMessage("Sent to first participant"));
Console.WriteLine("Message sent to the first participant queue.");

// Send a message to the second participant queue
await participant2Sender.SendMessageAsync(new ServiceBusMessage("Sent to second participant"));
Console.WriteLine("Message sent to the second participant queue.");

// Complete the transaction
transactionScope.Complete();
Console.WriteLine("Transaction completed successfully.");

// Prompt the user to press a key to exit
Console.WriteLine("Press any key to exit...");
Console.ReadKey(true);
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

```
using Azure.Messaging.ServiceBus;
using System.Transactions;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";
const string firstTransactionParticipantQueueName = "transactional-queues.participant-1";
const string secondTransactionParticipantQueueName = "transactional-queues.participant-2";

// Create a Service Bus client
ServiceBusClientOptions serviceBusClientOptions = new() { EnableCrossEntityTransactions = true };
await using ServiceBusClient client = new(connectionString, serviceBusClientOptions);

// Receive a message from the initiator queue
Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

// Create a TransactionScope object
using TransactionScope transactionScope = new(TransactionScopeAsyncFlowOption.Enabled);

// Complete the initiator message
await initiatorReceiver.CompleteMessageAsync(receivedMessage);

// Send a message to the first participant queue
await participant1Sender.SendMessageAsync(new ServiceBusMessage("Sent to first participant"));
Console.WriteLine("Message sent to the first participant queue.");

// Send a message to the second participant queue
await participant2Sender.SendMessageAsync(new ServiceBusMessage("Sent to second participant"));
Console.WriteLine("Message sent to the second participant queue.");

// Complete the transaction
transactionScope.Complete();
Console.WriteLine("Transaction completed successfully.");

// Prompt the user to press a key to exit
Console.WriteLine("Press any key to exit..");
Console.ReadKey(true);
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

```
using Azure.Messaging.ServiceBus;
using System.Transactions;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";
const string firstTransactionParticipantQueueName = "transactional-queues.participant-1";
const string secondTransactionParticipantQueueName = "transactional-queues.participant-2";

// Create a Service Bus client
ServiceBusClientOptions serviceBusClientOptions = new() { EnableCrossEntityTransactions = true };
await using ServiceBusClient client = new(connectionString, serviceBusClientOptions);

// Create a Service Bus receiver and senders
ServiceBusReceiver initiatorReceiver = client.CreateReceiver(initiatorQueueName);
ServiceBusSender participant1Sender = client.CreateSender(firstTransactionParticipantQueueName);
ServiceBusSender participant2Sender = client.CreateSender(secondTransactionParticipantQueueName);

// Receive a message from the initiator queue
Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

// Create a TransactionScope object
using TransactionScope transactionScope = new(TransactionScopeAsyncFlowOption.Enabled);

// Complete the initiator message
await initiatorReceiver.CompleteMessageAsync(receivedMessage);

Console.WriteLine("Message sent to the first participant queue.");

// Send a message to the second participant queue
await participant2Sender.SendMessageAsync(new ServiceBusMessage("Sent to second participant"));
Console.WriteLine("Message sent to the second participant queue.");

// Complete the transaction
transactionScope.Complete();
Console.WriteLine("Transaction completed successfully.");

// Prompt the user to press a key to exit
Console.WriteLine("Press any key to exit..");
Console.ReadKey(true);
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

```
using Azure.Messaging.ServiceBus;
using System.Transactions;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";
const string firstTransactionParticipantQueueName = "transactional-queues.participant-1";
const string secondTransactionParticipantQueueName = "transactional-queues.participant-2";

// Create a Service Bus client
ServiceBusClientOptions serviceBusClientOptions = new() { EnableCrossEntityTransactions = true };
await using ServiceBusClient client = new(connectionString, serviceBusClientOptions);

// Create a Service Bus receiver and senders
ServiceBusReceiver initiatorReceiver = client.CreateReceiver(initiatorQueueName);
ServiceBusSender participant1Sender = client.CreateSender(firstTransactionParticipantQueueName);
ServiceBusSender participant2Sender = client.CreateSender(secondTransactionParticipantQueueName);

// Receive a message from the initiator queue
Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

// Create a TransactionScope object
using TransactionScope transactionScope = new(TransactionScopeAsyncFlowOption.Enabled);

// Complete the initiator message
await initiatorReceiver.CompleteMessageAsync(receivedMessage);

// Send a message to the first participant queue
await participant1Sender.SendMessageAsync(new ServiceBusMessage("Sent to first participant"));
Console.WriteLine("Message sent to the first participant queue.");

// Send a message to the second participant queue
await participant2Sender.SendMessageAsync(new ServiceBusMessage("Sent to second participant"));
Console.WriteLine("Message sent to the second participant queue.");

// Prompt the user to press a key to exit
Console.WriteLine("Press any key to exit.. ");
Console.ReadKey(true);
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

```
using Azure.Messaging.ServiceBus;
using System.Transactions;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string initiatorQueueName = "transactional-queues.initiator";
const string firstTransactionParticipantQueueName = "transactional-queues.participant-1";
const string secondTransactionParticipantQueueName = "transactional-queues.participant-2";

// Create a Service Bus client
ServiceBusClientOptions serviceBusClientOptions = new() { EnableCrossEntityTransactions = true };
await using ServiceBusClient client = new(connectionString, serviceBusClientOptions);

// Create a Service Bus receiver and senders
ServiceBusReceiver initiatorReceiver = client.CreateReceiver(initiatorQueueName);
ServiceBusSender participant1Sender = client.CreateSender(firstTransactionParticipantQueueName);
ServiceBusSender participant2Sender = client.CreateSender(secondTransactionParticipantQueueName);

// Receive a message from the initiator queue
Console.WriteLine("Ready to receive messages from the initiator queue.");
ServiceBusReceivedMessage receivedMessage = await initiatorReceiver.ReceiveMessageAsync();
Console.WriteLine($"Received message: {receivedMessage.Body}");

// Create a TransactionScope object
using TransactionScope transactionScope = new(TransactionScopeAsyncFlowOption.Enabled);

// Complete the initiator message
await initiatorReceiver.CompleteMessageAsync(receivedMessage);

// Send a message to the first participant queue
await participant1Sender.SendMessageAsync(new ServiceBusMessage("Sent to first participant"));
Console.WriteLine("Message sent to the first participant queue.");

// Send a message to the second participant queue
await participant2Sender.SendMessageAsync(new ServiceBusMessage("Sent to second participant"));
Console.WriteLine("Message sent to the second participant queue.");

// Complete the transaction
transactionScope.Complete();
Console.WriteLine("Transaction completed successfully.");
```

# Demonstration

Create the Namespace

Create the Queues

Create the Initiator

Create Transaction Manager

Run the Demo

```
C:\Presentations\UnlockTheP... + ×
Press any key to start sending messages...
Message sent to the initiator queue.
Press any key to exit...
```

```
C:\Presentations\UnlockTheP... + ×
Ready to receive messages from the initiator queue.
Received message: Initiate Transaction
Message sent to the first participant queue.
Message sent to the second participant queue.
Transaction completed successfully.
Press any key to exit...
```

```
C:\Presentations\UnlockTheP... + ×
Transactional Participant 1 is listening for messages. Press any key to stop receiving messages...
Received message: Sent to first participant
```

```
C:\Presentations\UnlockTheP... + ×
Transactional Participant 2 is listening for messages. Press any key to stop receiving messages...
Received message: Sent to second participant
```

# Key Points to Remember

Atomic Operations

Consistency Checks

Resource Isolation

Rollback Mechanisms

# Key Points to Remember

- Aggregates related messages for batch processing and data consolidation.
- Enhances efficiency and scalability by reducing the overhead of processing individual messages.
- Provides a unified view of related information, simplifying analysis and decision-making.

# Claim Check Pattern

Advanced Processing Techniques

Unlock the Power of Messaging Patterns

# What is the Claim Check Pattern

Storing large payloads in external storage and passing a reference

Reduce the size of messages in the queue

# Key Components and Flow

External Storage  
System

Claim Check

Message Queue

Payload Retrieval  
Logic

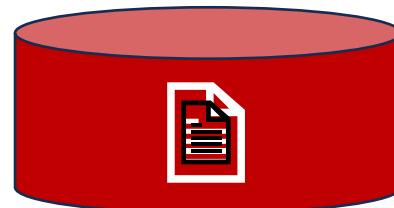
# Key Components and Flow

External Storage System

Claim Check

Message Queue

Payload Retrieval Logic



# Benefits

Improved  
Performance

Scalability

Resource  
Efficiency

# Drawbacks

Complexity

Latency

Dependencies

# Use Cases

Large File  
Transfers

Complex Data  
Structures

Resource  
Optimization

# Demonstration

Create the Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create the Namespace

Create the Queue

```
"Queues": [  
  {  
    "Name": "claim-check",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLetteredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": false  
    }  
  }]  
]
```

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

```
using Azure.Messaging.ServiceBus;
using Azure.Storage.Blobs;
using System.Text;

const string serviceBusConnectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageShared";
const string storageConnectionString = "UseDevelopmentStorage=true";
const string queueName = "claim-check";
const string blobContainerName = "claim-check";

// Prompt the user to start sending messages
Console.WriteLine("Press any key to start sending messages.");
Console.ReadKey(true);

// Create the payload
string payload = "This is a large payload that needs to be stored in external storage.";

// Create a unique identifier for the claim check
string claimCheckId = Guid.NewGuid().ToString();

// Create a blob client and upload the payload
BlobServiceClient blobServiceClient = new(storageConnectionString);
BlobContainerClient blobContainerClient = blobServiceClient.GetBlobContainerClient(blobContainerName);
await blobContainerClient.CreateIfNotExistsAsync();
BlobClient blobClient = blobContainerClient.GetBlobClient(claimCheckId);
using MemoryStream stream = new(Encoding.UTF8.GetBytes(payload));
await blobClient.UploadAsync(stream);

// Create and send the message
await using ServiceBusClient serviceBusClient = new(serviceBusConnectionString);
await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(queueName);
ServiceBusMessage message = new("Claim check for payload");
message.ApplicationProperties.Add("ClaimCheckId", claimCheckId);
await serviceBusSender.SendMessageAsync(message);
Console.WriteLine($"Message sent with claim check ID: {claimCheckId}");
```

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

Create the Consumer

```
using Azure.Messaging.ServiceBus;
using Azure.Storage.Blobs;
using System.Text;

const string serviceBusConnectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string storageConnectionString = "UseDevelopmentStorage=true";
const string queueName = "claim-check";
const string blobContainerName = "claim-check";

// Create a Service Bus client and processor
await using ServiceBusClient serviceBusClient = new(serviceBusConnectionString);
await using ServiceBusProcessor serviceBusProcessor = serviceBusClient.CreateProcessor(queueName);

// Register the message handler
serviceBusProcessor.ProcessMessageAsync += async processMessageEventArgs =>
{
    Console.WriteLine();

    // Get the claim check ID from the message
    string? claimCheckId = processMessageEventArgs.Message.ApplicationProperties["ClaimCheckId"].ToString();
    if (claimCheckId == null)
    {
        Console.WriteLine("Claim check ID not found in message.");
        await processMessageEventArgs.DeadLetterMessageAsync(processMessageEventArgs.Message);
        return;
    }
    Console.WriteLine($"Received message with claim check ID: {claimCheckId}");

    // Download the payload from the blob storage
    BlobServiceClient blobServiceClient = new(storageConnectionString);
    BlobContainerClient blobContainerClient = blobServiceClient.GetBlobContainerClient(blobContainerName);
    BlobClient blobClient = blobContainerClient.GetBlobClient(claimCheckId);
    using MemoryStream stream = new();
    await blobClient.DownloadToAsync(stream);
    string payload = Encoding.UTF8.GetString(stream.ToArray());

    // Process the payload
    Console.WriteLine($"Payload: {payload}");

    // Complete the message
    await processMessageEventArgs.CompleteMessageAsync(processMessageEventArgs.Message);
};

serviceBusProcessor.ProcessErrorAsync += processErrorEventArgs =>
{
    Console.WriteLine($"Exception: {processErrorEventArgs.Exception}");
    return Task.CompletedTask;
};

await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Press any key to stop processing messages.");
Console.ReadKey(true);

await serviceBusProcessor.StopProcessingAsync();
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

Create the Consumer

```
using Azure.Messaging.ServiceBus;
using Azure.Storage.Blobs;
using System.Text;

const string serviceBusConnectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string storageConnectionString = "UseDevelopmentStorage=true";
const string queueName = "claim-check";
const string blobContainerName = "claim-check";

// Get the claim check ID from the message
string? claimCheckId = processMessageEventArgs.Message.ApplicationProperties["ClaimCheckId"].ToString();
if (claimCheckId == null)
{
    Console.WriteLine("Claim check ID not found in message.");
    await processMessageEventArgs.DeadLetterMessageAsync(processMessageEventArgs.Message);
    return;
}

Console.WriteLine($"Received message with claim check ID: {claimCheckId}");

// Download the payload from the blob storage
BlobServiceClient blobServiceClient = new(storageConnectionString);
BlobContainerClient blobContainerClient = blobServiceClient.GetBlobContainerClient(blobContainerName);
BlobClient blobClient = blobContainerClient.GetBlobClient(claimCheckId);
using MemoryStream stream = new();
await blobClient.DownloadToAsync(stream);
string payload = Encoding.UTF8.GetString(stream.ToArray());

// Process the payload
Console.WriteLine($"Payload: {payload}");

// Complete the message
await processMessageEventArgs.CompleteMessageAsync(processMessageEventArgs.Message);

};

await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Press any key to stop processing messages.");
Console.ReadKey(true);

await serviceBusProcessor.StopProcessingAsync();
```

# Demonstration

Create the Namespace

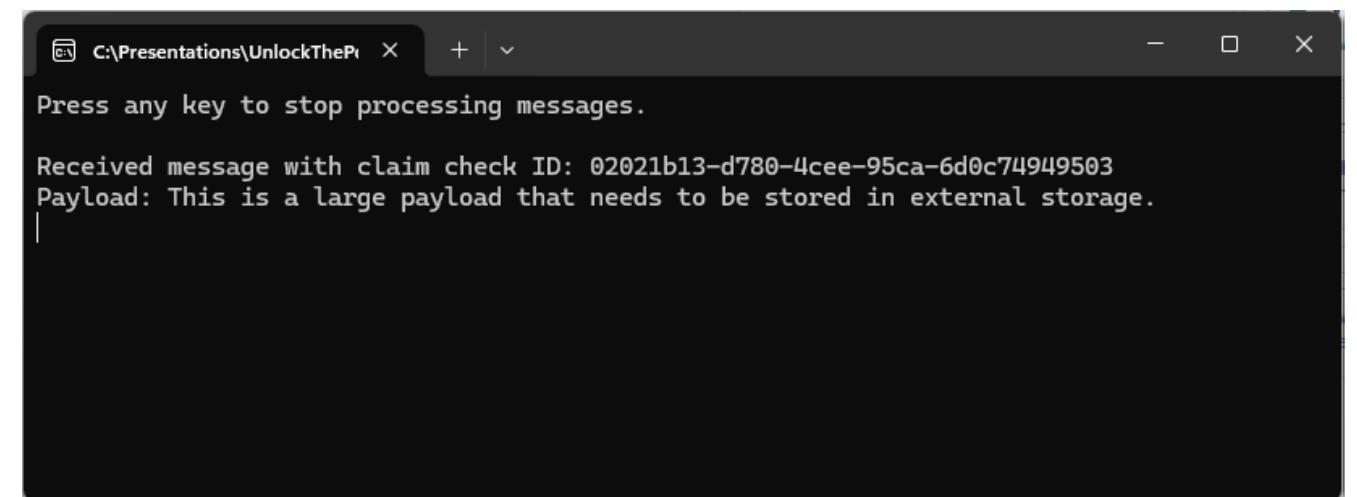
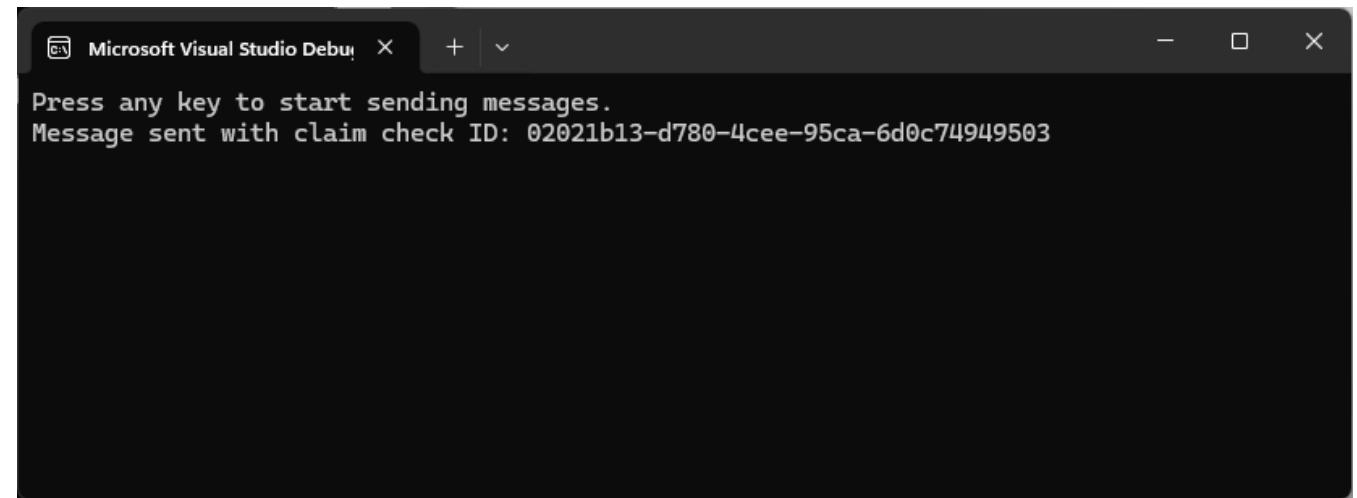
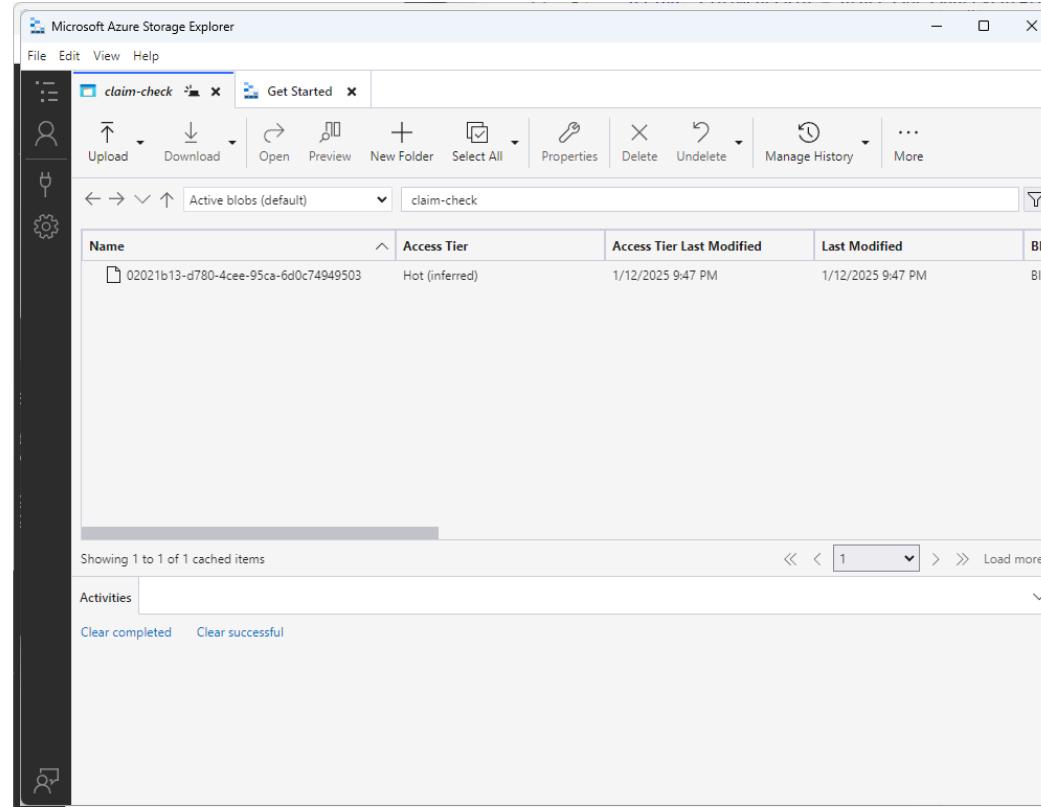
Create the Queue

Create the Producer

Create the Consumer

Run the Demo

# Demonstration



# Key Points to Remember

Store Large Payloads Externally

Include Claim Checks

Ensure Storage Reliability

Coordinate Components

# Wrap-Up

Advanced Processing Techniques

Unlock the Power of Messaging Patterns

# Key Takeaways

Idempotent  
Receivers

Message  
Scheduling

Transactional  
Queues

Claim Checks

# Preview: Resilience and Reliability

Circuit Breaker

Saga

Sequential Convoy

# Resilience and Reliability

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Circuit Breaker

Resilience and Reliability

Unlock the Power of Messaging Patterns

# What is the Circuit Breaker Pattern

Detects and handles  
failures gracefully

Improve system resilience  
and stability

# Key Components and Flow

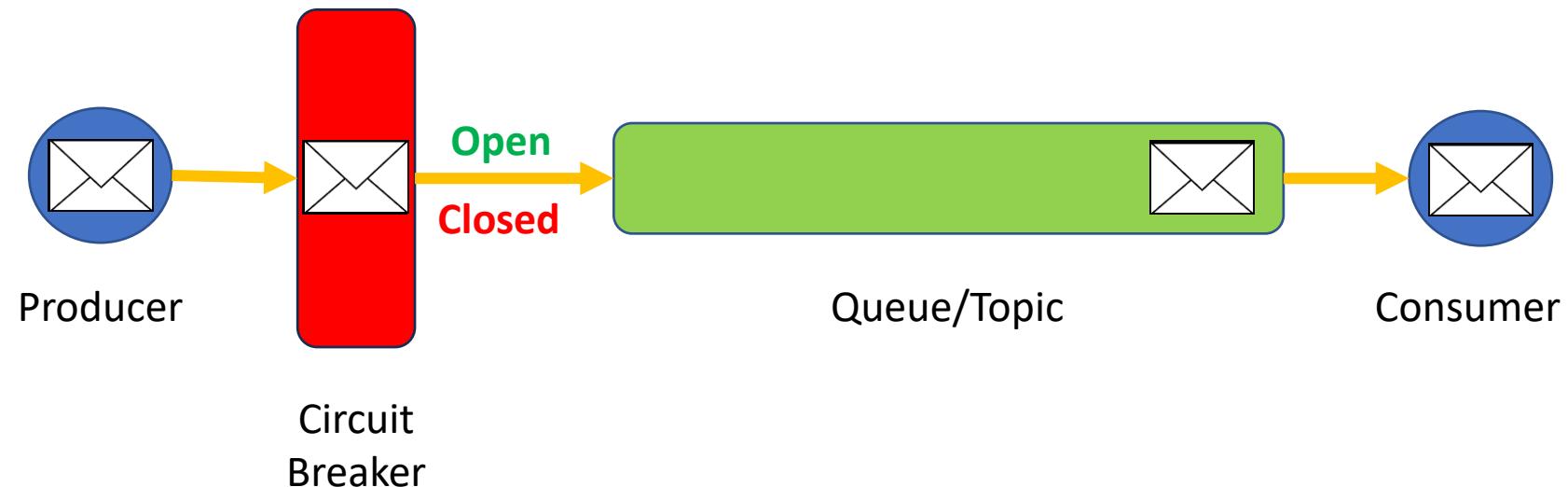
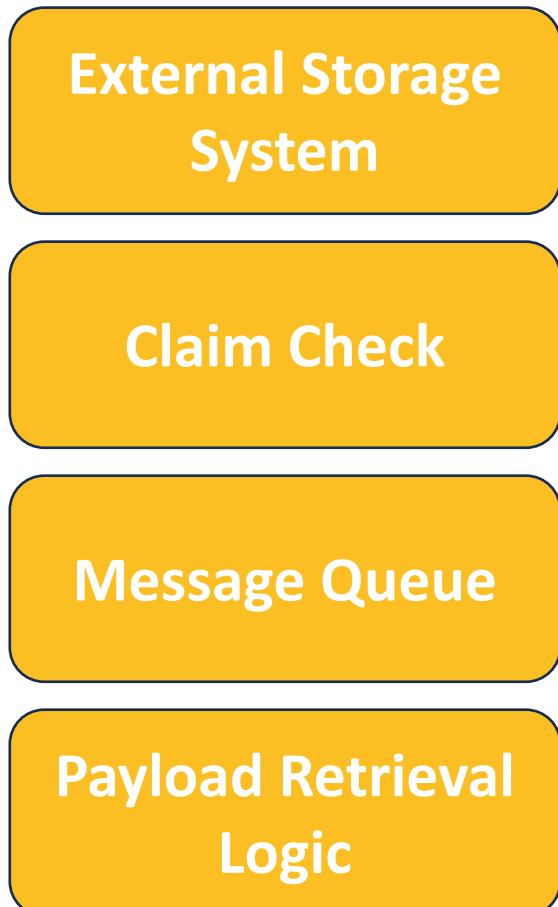
Circuit Breaker  
State

Failure Threshold

Fallback  
Mechanism

Recovery Logic

# Key Components and Flow



# Benefits

Resiliency

Faster Failure  
Detection

Resource  
Optimization

# Drawbacks

Complexity

Latency

False Positives

# Use Cases

External Service  
Failures

Temporary  
Overload

Error Threshold  
Management

# Demonstration

Create the Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create the Namespace

Create the Queue

```
"Queues": [
  {
    "Name": "circuit-breaker",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadLetteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

```
using Azure.Messaging.ServiceBus;
using Polly;
using Polly.CircuitBreaker;
using System.Text;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=SAS_KEY_VALUE;UseDevelopmentEmulator=true;";
const string topicName = "circuit-breaker";

int delayInBetweenMessages = 100;

AsyncCircuitBreakerPolicy circuitBreakerPolicy = Policy
    .Handle<Exception>()
    .CircuitBreakerAsync(
        exceptionsAllowedBeforeBreaking: 3,
        durationOfBreak: TimeSpan.FromSeconds(5),
        onBreak: (exception, breakDelay) =>
    {
        Console.WriteLine($"Circuit breaker tripped! Breaking for {breakDelay.TotalSeconds} seconds due to: {exception.Message}");
        delayInBetweenMessages = 5000;
    },
    onReset: () =>
    {
        Console.WriteLine("Circuit breaker reset.");
        delayInBetweenMessages = 100;
    },
    onHalfOpen: () =>
    {
        Console.WriteLine("Circuit breaker is half-open. Testing the operation...");
    });
}

for (int i = 0; i < 30; i++)
{
    await SendMessageWithCircuitBreakerAsync();
    Thread.Sleep(delayInBetweenMessages);
}

Console.WriteLine();
Console.WriteLine("Done sending messages; press any key to exit...");
Console.ReadKey();
```

1 reference | authors: changes  
async Task SendMessageWithCircuitBreakerAsync()  
{  
  
 await using ServiceBusClient serviceBusClient = new(connectionString);  
 await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(topicName);  
  
 try
 {
 await circuitBreakerPolicy.ExecuteAsync(async () =>
 {
 string messageBody = "This is a test message for the Circuit Breaker pattern.";
 ServiceBusMessage message = new(Encoding.UTF8.GetBytes(messageBody));  
  
 // Simulating a failure scenario
 if (DateTime.UtcNow.Second % 2 == 0)
 {
 throw new Exception("Simulated failure.");
 }
  
 await serviceBusSender.SendMessageAsync(message);
 Console.WriteLine(\$"Sent a message at: {DateTime.UtcNow}");
 });
 }
 catch (Exception ex)
 {
 Console.WriteLine(\$"Attempted to send a message at: {DateTime.UtcNow}");
 }
}

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

```
using Azure.Messaging.ServiceBus;  
  
const string connectionString = "Endpoint=sb://  
const string topicName = "circuit-breaker";  
  
int delayInBetweenMessages = 100;  
  
delayInBetweenMessages = 5000;  
onReset: () =>  
{  
    Console.WriteLine("Circuit breaker reset.");  
    delayInBetweenMessages = 100;  
},  
onHalfOpen: () =>  
{  
    Console.WriteLine("Circuit breaker is half-open. Testing the operation...");  
});  
  
for (int i = 0; i < 30; i++)  
{  
    await SendMessageWithCircuitBreakerAsync();  
    Thread.Sleep(delayInBetweenMessages);  
}  
  
Console.WriteLine();  
Console.WriteLine("Done sending messages, press any key to exit...");  
Console.ReadKey();  
  
[Reference: https://github.com/Azure/azure-sdk-for-net/tree/main/sdk/servicebus/Azure.Messaging.ServiceBus  
async Task SendMessageWithCircuitBreakerAsync()  
{  
  
    await using ServiceBusClient serviceBusClient = new(connectionString);  
    await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(topicName);  
  
    try  
    {  
        await circuitBreakerPolicy.ExecuteAsync(async () =>  
        {  
            string messageBody = "This is a test message for the Circuit Breaker pattern.";  
            ServiceBusMessage message = new(Encoding.UTF8.GetBytes(messageBody));  
  
            // Simulating a failure scenario  
            if (DateTime.UtcNow.Second % 2 == 0)  
            {  
                throw new Exception("Simulated failure.");  
            }  
  
            await serviceBusSender.SendMessageAsync(message);  
            Console.WriteLine($"Sent a message at: {DateTime.UtcNow}");  
        });  
    }  
    catch (Exception ex)  
    {  
        Console.WriteLine($"Attempted to send a message at: {DateTime.UtcNow}");  
    }  
}
```

Unlock the Power of Messaging Patterns

# Demonstration

## Create the Namespace

```
using Azure.Messaging.ServiceBus;
using Polly;
using Polly.CircuitBreaker;
using System.Text;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=SAS_KEY_VALUE;UseDevelopmentEmulator=true";
const string topicName = "circuit-breaker";

AsyncCircuitBreakerPolicy circuitBreakerPolicy = Policy
    .Handle<Exception>()
    .CircuitBreakerAsync(
        exceptionsAllowedBeforeBreaking: 3,
        durationOfBreak: TimeSpan.FromSeconds(5),
        onBreak: (exception, breakDelay) =>
    {
        Console.WriteLine($"Circuit breaker tripped! Breaking for {breakDelay.TotalSeconds} seconds due to: {exception.Message}");
        delayInBetweenMessages = 5000;
    },
    onReset: () =>
    {
        Console.WriteLine("Circuit breaker reset.");
        delayInBetweenMessages = 100;
    },
    onHalfOpen: () =>
    {
        Console.WriteLine("Circuit breaker is half-open. Testing the operation...");
    });

```

```
    });
}
catch (Exception ex)
{
    Console.WriteLine($"Attempted to send a message at: {DateTime.UtcNow}");
}
}
```

# Demonstration

# Create the Producer

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

```
using Azure.Messaging.ServiceBus;
using Polly;
using Polly.CircuitBreaker;
using System.Text;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=SAS_KEY_VALUE;UseDevelopmentEmulator=true";

for (int i = 0; i < 30; i++)
{
    await SendMessageWithCircuitBreakerAsync();
    Thread.Sleep(delayInBetweenMessages);
}

Console.WriteLine();
Console.WriteLine("Done sending messages; press any key to exit...");
Console.ReadKey();
```

```
using Azure.Messaging.ServiceBus;
using Polly;
using Polly.CircuitBreaker;
using System.Text;

async Task SendMessageWithCircuitBreaker()
{
    await using ServiceBusClient serviceBusClient = new(connectionString);
    await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(topicName);

    try
    {
        await circuitBreakerPolicy.ExecuteAsync(async () =>
        {
            string messageBody = "This is a test message for the Circuit Breaker pattern.";
            ServiceBusMessage message = new(Encoding.UTF8.GetBytes(messageBody));

            // Simulating a failure scenario
            if (DateTime.UtcNow.Second % 2 == 0)
            {
                throw new Exception("Simulated failure.");
            }

            await serviceBusSender.SendMessageAsync(message);
            Console.WriteLine($"Sent a message at: {DateTime.UtcNow}");
        });
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Attempted to send a message at: {DateTime.UtcNow}");
    }
}
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

Run the Demo

```
C:\Presentations\UnlockTheP... X + - □ ×  
Press any key to start sending messages...  
  
Sent a message at: 1/13/2025 12:51:55 PM  
Sent a message at: 1/13/2025 12:51:55 PM  
Attempted to send a message at: 1/13/2025 12:51:56 PM  
Attempted to send a message at: 1/13/2025 12:51:56 PM  
Circuit breaker tripped! Breaking for 5 seconds due to: Simulated failure.  
Attempted to send a message at: 1/13/2025 12:51:56 PM  
Circuit breaker is half-open. Testing the operation...  
Sent a message at: 1/13/2025 12:52:01 PM  
Circuit breaker reset.  
Sent a message at: 1/13/2025 12:52:01 PM  
Sent a message at: 1/13/2025 12:52:01 PM  
Sent a message at: 1/13/2025 12:52:02 PM  
Attempted to send a message at: 1/13/2025 12:52:02 PM  
Attempted to send a message at: 1/13/2025 12:52:02 PM  
Circuit breaker tripped! Breaking for 5 seconds due to: Simulated failure.  
Attempted to send a message at: 1/13/2025 12:52:02 PM  
Circuit breaker is half-open. Testing the operation...  
Sent a message at: 1/13/2025 12:52:07 PM  
Circuit breaker reset.  
Sent a message at: 1/13/2025 12:52:07 PM  
Attempted to send a message at: 1/13/2025 12:52:08 PM  
Attempted to send a message at: 1/13/2025 12:52:08 PM  
Circuit breaker tripped! Breaking for 5 seconds due to: Simulated failure.  
Attempted to send a message at: 1/13/2025 12:52:08 PM  
Circuit breaker is half-open. Testing the operation...  
Sent a message at: 1/13/2025 12:52:13 PM  
Circuit breaker reset.  
Sent a message at: 1/13/2025 12:52:13 PM  
Sent a message at: 1/13/2025 12:52:13 PM  
Attempted to send a message at: 1/13/2025 12:52:14 PM  
Attempted to send a message at: 1/13/2025 12:52:14 PM  
Circuit breaker tripped! Breaking for 5 seconds due to: Simulated failure.  
Attempted to send a message at: 1/13/2025 12:52:14 PM  
Circuit breaker is half-open. Testing the operation...  
Sent a message at: 1/13/2025 12:52:19 PM  
Circuit breaker reset.  
Sent a message at: 1/13/2025 12:52:19 PM  
Sent a message at: 1/13/2025 12:52:20 PM  
Attempted to send a message at: 1/13/2025 12:52:20 PM  
Attempted to send a message at: 1/13/2025 12:52:20 PM  
Circuit breaker tripped! Breaking for 5 seconds due to: Simulated failure.  
Attempted to send a message at: 1/13/2025 12:52:20 PM  
Circuit breaker is half-open. Testing the operation...  
Sent a message at: 1/13/2025 12:52:25 PM  
Circuit breaker reset.  
  
Done sending messages; press any key to exit...
```

# Key Points to Remember

Configure Thresholds

Implement Fallbacks

Monitor and Adjust

Test Recovery

# Saga Pattern

Resilience and Reliability

Unlock the Power of Messaging Patterns

# What is the Circuit Breaker Pattern

Breaks the transaction  
into a series of smaller,  
independent steps

Managing long-running  
transactions

# Key Components and Flow

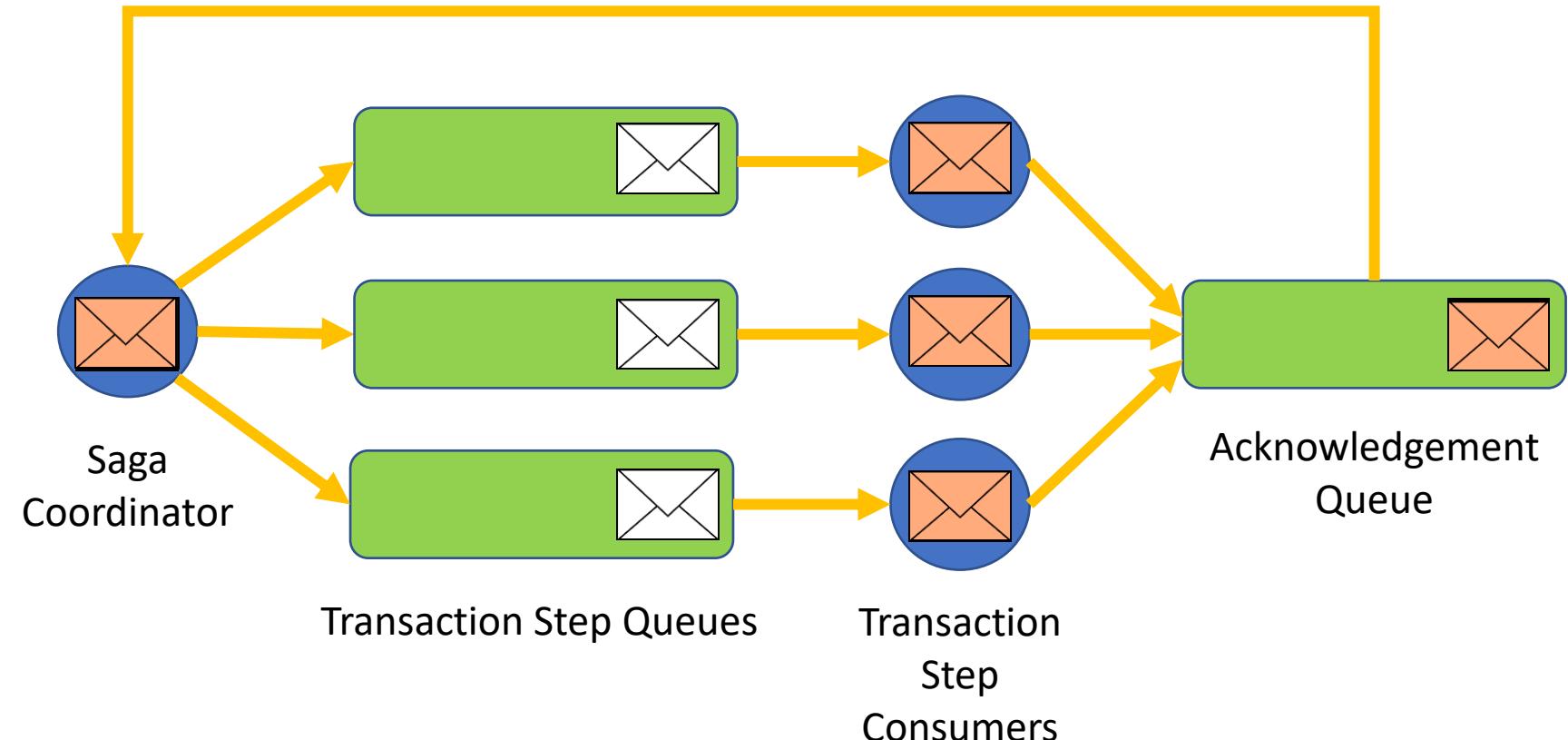
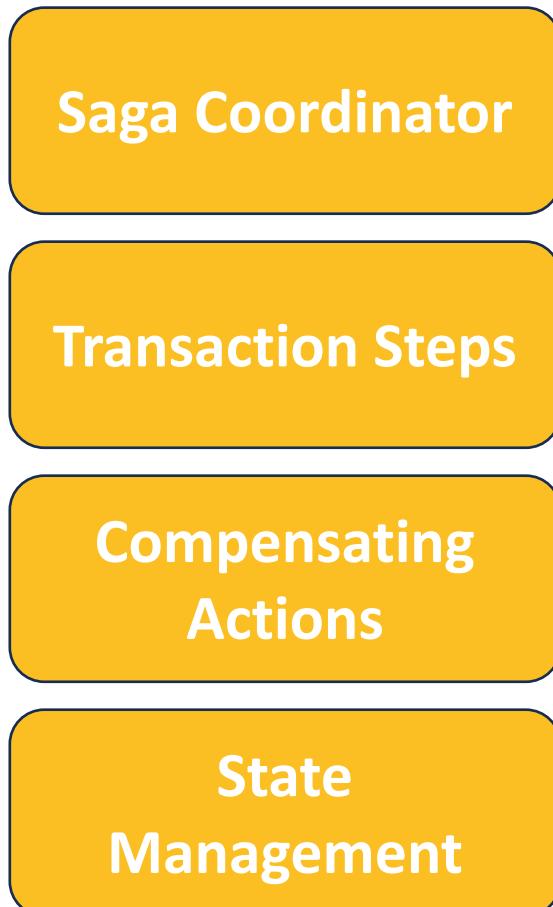
Saga Coordinator

Transaction Steps

Compensating  
Actions

State  
Management

# Key Components and Flow



# Benefits

Data Consistency

Fault Tolerance

Scalability

# Drawbacks

Complexity

Compensating  
Logic

Latency

# Use Cases

Microservices  
Architecture

E-Commerce  
Order Processing

Travel Booking

# Demonstration

Create the Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create the Namespace

Create the Queues

```
"Queues": [
  {
    "Name": "saga.order",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadletteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "saga.payment",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadletteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "saga.inventory",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadletteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  },
  {
    "Name": "saga.completion",
    "Properties": {
      "DeadLetteringOnMessageExpiration": false,
      "DefaultMessageTimeToLive": "PT1H",
      "DuplicateDetectionHistoryTimeWindow": "PT20S",
      "ForwardDeadletteredMessagesTo": "",
      "ForwardTo": "",
      "LockDuration": "PT1M",
      "MaxDeliveryCount": 3,
      "RequiresDuplicateDetection": false,
      "RequiresSession": false
    }
  }
]
```

saga.order  
saga.payment  
saga.inventory  
saga.completion

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

```
namespace SagaCoordinator;

2 references | Chad Green, 8 minutes ago | 1 author, 1 change
public class SagaAcknowledgements
{
    2 references | Chad Green, 8 minutes ago | 1 author, 1 change
    public bool OrderAcknowledged { get; set; }
    2 references | Chad Green, 8 minutes ago | 1 author, 1 change
    public bool PaymentAcknowledged { get; set; }
    2 references | Chad Green, 8 minutes ago | 1 author, 1 change
    public bool InventoryAcknowledged { get; set; }
    2 references | Chad Green, 8 minutes ago | 1 author, 1 change
    public bool SagaIsComplete => OrderAcknowledged && PaymentAcknowledged && InventoryAcknowledged;
}
```

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

```
using Azure.Messaging.ServiceBus;
using System.Text;
using System;

namespace SagaCoordinator
{
    // references | Chet Green, 9 minutes ago | 1 author, 1 change
    public class SagaCoordinator
    {
        ServiceBusClient serviceBusClient,
        string orderQueueName,
        string paymentQueueName,
        string inventoryQueueName,
        string completionQueueName,
        string compensationQueueName
    }

    private readonly ServiceBusClient _serviceBusClient = serviceBusClient;
    private readonly string _orderQueueName = orderQueueName;
    private readonly string _paymentQueueName = paymentQueueName;
    private readonly string _inventoryQueueName = inventoryQueueName;
    private readonly string _completionQueueName = completionQueueName;
    private readonly string _compensationQueueName = compensationQueueName

    // references | Chet Green, 9 minutes ago | 1 author, 1 change
    public async Task StartSagaAsync(string orderId)
    {
        try
        {
            SagaAcknowledgments acknowledgments = new();
            Console.WriteLine($"Saga for {orderId} has been started");

            await SendMessageAsync(_orderQueueName, $"Create Order for {orderId}");
            await SendMessageAsync(_paymentQueueName, $"Charge Payment for {orderId}");
            await SendMessageAsync(_inventoryQueueName, $"Update Inventory for {orderId}");
            //Console.WriteLine($"Saga for {orderId} has been completed");

            // Wait for the acknowledgments
            await UsingServiceBusReceiver(serviceBusReceiver = _serviceBusClient.CreateReceiver(_completionQueueName));
            while (!acknowledgments.SagaIsComplete)
            {
                ServiceBusReceivedMessage receivedMessage = await serviceBusReceiver.ReceiveMessageAsync();
                if (receivedMessage is null)
                    continue;

                string? serviceName = receivedMessage.ApplicationProperties["Service"].ToString();
                if (serviceName is not null)
                {
                    bool success = (bool)receivedMessage.ApplicationProperties["Success"];
                    Console.WriteLine($"Received acknowledgement from {serviceName}: {success ? "Success" : "Failure"}");
                    if (success)
                        await CompensateAsync(orderId);
                    Console.WriteLine($"Saga for {orderId} has been aborted due to an error in {serviceName}");
                    break;
                }
                switch (serviceName)
                {
                    case "Order":
                        acknowledgments.OrderAcknowledged = true;
                        break;
                    case "Payment":
                        acknowledgments.PaymentAcknowledged = true;
                        break;
                    case "Inventory":
                        acknowledgments.InventoryAcknowledged = true;
                        break;
                }
                await serviceBusReceiver.CompleteMessageAsync(receivedMessage);
            }
        }
        catch (Exception ex)
        {
            await CompensateAsync(orderId);
            Console.WriteLine($"Saga for {orderId} has been aborted due to an error: {ex.Message}");
        }
    }

    // references | Chet Green, 10 minutes ago | 1 author, 1 change
    private async Task SendMessageAsync(string queueName, string message)
    {
        await UsingServiceBusSender(serviceBusSender = _serviceBusClient.CreateSender(queueName));
        await serviceBusSender.SendMessageAsync(new ServiceBusMessage(Encoding.UTF8.GetBytes(message)));
        Console.WriteLine($"Message sent to {queueName}: {message}");
    }

    // references | Chet Green, 10 minutes ago | 1 author, 1 change
    private async Task CompensateAsync(string orderId)
    {
        await SendMessageAsync(_inventoryQueueName, $"Revert Inventory for {orderId}");
        await SendMessageAsync(_paymentQueueName, $"Refund Payment for {orderId}");
        await SendMessageAsync(_orderQueueName, $"Cancel Order for {orderId}");
        Console.WriteLine($"Saga for {orderId} has compensated");
    }
}
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

```
using Azure.Messaging.ServiceBus;
using System.Text;

namespace SagaCoordinator
{
    public class SagaCoordinator
    {
        private readonly ServiceBusClient serviceBusClient;
        string orderQueueName;
        string paymentQueueName;
        string inventoryQueueName;
        string completionQueueName;

        public SagaCoordinator(
            ServiceBusClient serviceBusClient,
            string orderQueueName,
            string paymentQueueName,
            string inventoryQueueName,
            string completionQueueName)
        {

            private readonly ServiceBusClient _serviceBusClient = serviceBusClient;
            private readonly string _orderQueueName = orderQueueName;
            private readonly string _paymentQueueName = paymentQueueName;
            private readonly string _inventoryQueueName = inventoryQueueName;
            private readonly string _completionQueueName = completionQueueName;

            if (Console.KeyAvailable)
            {
                if (Console.ReadKey(true).KeyChar == 'q')
                {
                    await StopSaga();
                }
            }
        }

        public async Task StartSagaAsync(string orderId)...
        {
            try
            {
                var message = new Message("Hello, World!");
                await serviceBusClient.CreateOrReplaceMessageAsync(_orderQueueName, message);
            }
            catch (Exception ex)
            {
                await Console.Error.WriteLineAsync(ex.Message);
            }
        }

        private async Task SendMessageAsync(string queueName, string message)...
        {
            var message = new Message("Hello, World!");
            await serviceBusClient.CreateOrReplaceMessageAsync(queueName, message);
        }

        private async Task CompensateAsync(string orderId)...
        {
            var message = new Message("Hello, World!");
            await serviceBusClient.CreateOrReplaceMessageAsync(_completionQueueName, message);
        }
    }
}
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

```
using Azure.Messaging.ServiceBus;
using System.Text;
namespace SagaCoordinator;
// references | Chad Green, 11 minutes ago | 1 author, 1 change
public class SagaCoordinator : ISagaCoordinator
{
    private readonly ServiceBusClient _serviceBusClient = serviceBusClient;
    private readonly string _orderQueueName;
    private readonly string _paymentQueueName;
    private readonly string _inventoryQueueName;
    private readonly string _completionQueueName;
}
```

6 references | Chad Green, 11 minutes ago | 1 author, 1 change  
**private async Task SendMessageAsync(string queueName, string message)**

```
{
    await using ServiceBusSender serviceBusSender = _serviceBusClient.CreateSender(queueName);
    await serviceBusSender.SendMessageAsync(new ServiceBusMessage(Encoding.UTF8.GetBytes(message)));
    Console.WriteLine($"Message sent to {queueName}: {message}");
}
```

2 references | Chad Green, 11 minutes ago | 1 author, 1 change  
**private async Task CompenstateAsync(string orderId)**

```
{
    await SendMessageAsync(_inventoryQueueName, $"Revert Inventory for {orderId}");
    await SendMessageAsync(_paymentQueueName, $"Refund Payment for {orderId}");
    await SendMessageAsync(_orderQueueName, $"Cancel Order for {orderId}");
    Console.WriteLine($"Saga for {orderId} has been compensated");
}
```

```
private async Task SendMessageAsync(string queueName, string message)
{
    await using ServiceBusSender serviceBusSender = _serviceBusClient.CreateSender(queueName);
    await serviceBusSender.SendMessageAsync(new ServiceBusMessage(Encoding.UTF8.GetBytes(message)));
    Console.WriteLine($"Message sent to {queueName}: {message}");
}

// references | Chad Green, 11 minutes ago | 1 author, 1 change
private async Task CompenstateAsync(string orderId)
{
    await SendMessageAsync(_inventoryQueueName, $"Revert Inventory for {orderId}");
    await SendMessageAsync(_paymentQueueName, $"Refund Payment for {orderId}");
    await SendMessageAsync(_orderQueueName, $"Cancel Order for {orderId}");
    Console.WriteLine($"Saga for {orderId} has been compensated");
}
```

# Demonstration

# Create the Saga Coordinator

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

Create the Services

```
using Azure.Messaging.ServiceBus;
using System.Text;

namespace Core;

3 references | Chad Green, 14 minutes ago | 1 author, 1 change
public static class ServiceHelper
{
    3 references | Chad Green, 14 minutes ago | 1 author, 1 change
    public static async Task SendAcknowledgementAsync(
        ServiceBusClient serviceBusClient,
        string completionQueueName,
        string service,
        bool success)
    {
        await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(completionQueueName);
        ServiceBusMessage serviceBusMessage = new(Encoding.UTF8.GetBytes("Acknowlwedgement"));
        serviceBusMessage.ApplicationProperties.Add("Service", service);
        serviceBusMessage.ApplicationProperties.Add("Success", success);
        await serviceBusSender.SendMessageAsync(serviceBusMessage);
        Console.WriteLine($"Sent Acknowledgement for {service} : {(success ? "Success" : "Failure")}");
    }
}
```

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

Create the Services

```
using Azure.Messaging.ServiceBus;
using Core;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string inventoryQueueName = "saga.inventory";
const string completionQueueName = "saga.completion";

await using ServiceBusClient client = new(connectionString);
await using ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(inventoryQueueName);

// Register the message handler
serviceBusProcessor.ProcessMessageAsync += async args =>
{
    Console.WriteLine();
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Updating inventory: {body}");
    // Update the inventory
    await args.CompleteMessageAsync(args.Message);
    await ServiceHelper.SendAcknowledgementAsync(client, completionQueueName, "Inventory", true);
};

// Register the error handler
serviceBusProcessor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing
await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Processing Inventory messages. Press any key to exit...");
Console.ReadKey(true);

// Stop processing
await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine("Stopped processing Inventory messages.");
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

Create the Services

```
using Azure.Messaging.ServiceBus;
using Core;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string orderQueueName = "saga.order";
const string completionQueueName = "saga.completion";

await using ServiceBusClient client = new(connectionString);
await using ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(orderQueueName);

// Register the message handler
serviceBusProcessor.ProcessMessageAsync += async args =>
{
    Console.WriteLine();
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Processing order: {body}");
    // Process the order
    await args.CompleteMessageAsync(args.Message);
    await ServiceHelper.SendAcknowledgementAsync(client, completionQueueName, "Order", true);
};

// Register the error handler
serviceBusProcessor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing
await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Processing Order messages. Press any key to exit...");
Console.ReadKey(true);

// Stop processing
await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine("Stopped processing Order messages.");
```

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

Create the Services

```
using Azure.Messaging.ServiceBus;
using Core;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string paymentQueueName = "saga.payment";
const string completionQueueName = "saga.completion";

await using ServiceBusClient client = new(connectionString);
await using ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(paymentQueueName);

// Register the message handler
serviceBusProcessor.ProcessMessageAsync += async args =>
{
    Console.WriteLine();
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Processing payment: {body}");
    // Process the payment
    await args.CompleteMessageAsync(args.Message);
    await ServiceHelper.SendAcknowledgementAsync(client, completionQueueName, "Payment", true);
};

// Register the error handler
serviceBusProcessor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing
await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Processing Payment messages. Press any key to exit...");
Console.ReadKey(true);

// Stop processing
await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine("Stopped processing Payment messages.");
```

# Demonstration

## Create the Namespace

```
C:\Presentations\UnlockThePowerOfMessaging> Press any key to start the saga...
Saga for 123 has been started
Message sent to saga.order: Create Order for 123
Message sent to saga.payment: Charge Payment for 123
Message sent to saga.inventory: Update Inventory for 123
Received acknowledgement from Order: Success
Received acknowledgement from Payment: Success
Received acknowledgement from Inventory: Success
Saga for 123 has been completed

C:\Presentations\UnlockThePowerOfMessaging> Press any key to exit...
```

```
C:\Presentations\UnlockThePowerOfMessaging> Processing Payment messages. Press any key to exit...
Processing payment: Charge Payment for 123
Sent Acknowledgement for Payment : Success

C:\Presentations\UnlockThePowerOfMessaging> Processing Order messages. Press any key to exit...
Processing order: Create Order for 123
Sent Acknowledgement for Order : Success

C:\Presentations\UnlockThePowerOfMessaging> Processing Inventory messages. Press any key to exit...
Updating inventory: Update Inventory for 123
Sent Acknowledgement for Inventory : Success
```

# Key Points to Remember

Define Compensating Actions

Coordinate Steps

Track State

Handle Failures Gracefully

# Sequence Convoy Pattern

Resilience and Reliability

Unlock the Power of Messaging Patterns

# What is the Circuit Breaker Pattern

Ensures that related messages are processed in order

Enforce message ordering

# Key Components and Flow

Message Queue

Order Key

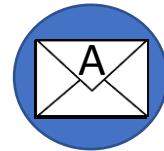
Sequential  
Processor

# Key Components and Flow

Message Queue

Order Key

Sequential  
Processor



# Benefits

Data Integrity

Consistency

Reliability

# Drawbacks

Reduced  
Throughput

Complexity

Potential Latency

# Use Cases

Order Processing

Event Sourcing

Workflow  
Management

# Demonstration

Create the Namespace

```
{  
  "UserConfig": {  
    "Namespaces": [  
      {  
        "Name": "sbemulators",  
        "Queues": [],  
        "Topics": [],  
        "Logging": {  
          "Type": "File"  
        }  
      }  
    ]  
  }  
}
```

# Demonstration

Create the Namespace

Create the Queue

```
"Queues": [  
  {  
    "Name": "sequential-convoy",  
    "Properties": {  
      "DeadLetteringOnMessageExpiration": false,  
      "DefaultMessageTimeToLive": "PT1H",  
      "DuplicateDetectionHistoryTimeWindow": "PT20S",  
      "ForwardDeadLetteredMessagesTo": "",  
      "ForwardTo": "",  
      "LockDuration": "PT1M",  
      "MaxDeliveryCount": 3,  
      "RequiresDuplicateDetection": false,  
      "RequiresSession": true  
    }  
  }  
]
```

# Demonstration

Create the Namespace

Create the Queues

Create the Producer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string queueName = "sequential-convoy";

// Prompt the user to start sending messages
Console.WriteLine("Press any key to start sending messages...");
Console.ReadKey(true);

// Build the list of messages to send
List<ServiceBusMessage> messages = [];
messages.Add(new ServiceBusMessage("Message 1 for order order-1") { SessionId = "order-1" });
messages.Add(new ServiceBusMessage("Message 1 for order order-2") { SessionId = "order-2" });
messages.Add(new ServiceBusMessage("Message 2 for order order-2") { SessionId = "order-2" });
messages.Add(new ServiceBusMessage("Message 3 for order order-2") { SessionId = "order-2" });
messages.Add(new ServiceBusMessage("Message 2 for order order-1") { SessionId = "order-1" });
messages.Add(new ServiceBusMessage("Message 3 for order order-1") { SessionId = "order-1" });

// Create a Service Bus client and sender
await using ServiceBusClient serviceBusClient = new(connectionString);
await using ServiceBusSender serviceBusSender = serviceBusClient.CreateSender(queueName);

// Send the messages to the Service Bus queue
foreach (ServiceBusMessage message in messages)
{
    await serviceBusSender.SendMessageAsync(message);
    Console.WriteLine($"Sent message: {message.Body}");
}
```

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

Create the Consumer

```
using Azure.Messaging.ServiceBus;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKe
const string queueName = "sequential-convoy";

Console.WriteLine("Waiting for messages...");

await using ServiceBusClient serviceBusClient = new(connectionString);

while (true)
{
    await using ServiceBusSessionReceiver sessionReceiver = await serviceBusClient.AcceptNextSessionAsync(queueName);
    if (sessionReceiver == null)
    {
        break; // No more sessions available
    }

    Console.WriteLine();
    Console.WriteLine($"Accepted session: {sessionReceiver.SessionId}");

    ServiceBusReceivedMessage message = await sessionReceiver.ReceiveMessageAsync();
    while (message != null)
    {
        string body = message.Body.ToString();
        string sessionId = message.SessionId;
        Console.WriteLine($"Received message: {body}");
        await sessionReceiver.CompleteMessageAsync(message);
        message = await sessionReceiver.ReceiveMessageAsync(TimeSpan.FromSeconds(2));
    }
}
```

# Demonstration

Create the Namespace

Create the Queues

Create the Saga Coordinator

Create the Services

```
using Azure.Messaging.ServiceBus;
using Core;

const string connectionString = "Endpoint=sb://127.0.0.1;SharedAccessKeyName=RootManageSharedAccessKey";
const string inventoryQueueName = "saga.inventory";
const string completionQueueName = "saga.completion";

await using ServiceBusClient client = new(connectionString);
await using ServiceBusProcessor serviceBusProcessor = client.CreateProcessor(inventoryQueueName);

// Register the message handler
serviceBusProcessor.ProcessMessageAsync += async args =>
{
    Console.WriteLine();
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Updating inventory: {body}");
    // Update the inventory
    await args.CompleteMessageAsync(args.Message);
    await ServiceHelper.SendAcknowledgementAsync(client, completionQueueName, "Inventory", true);
};

// Register the error handler
serviceBusProcessor.ProcessErrorAsync += args =>
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
};

// Start processing
await serviceBusProcessor.StartProcessingAsync();
Console.WriteLine("Processing Inventory messages. Press any key to exit...");
Console.ReadKey(true);

// Stop processing
await serviceBusProcessor.StopProcessingAsync();
Console.WriteLine("Stopped processing Inventory messages.");
```

Unlock the Power of Messaging Patterns

# Demonstration

Create the Namespace

Create the Queue

Create the Producer

Create the Consumer

Run the Demo

```
Microsoft Visual Studio Debug X + | v - □ ×
Press any key to start sending messages...
Sent message: Message 1 for order order-1
Sent message: Message 1 for order order-2
Sent message: Message 2 for order order-2
Sent message: Message 3 for order order-2
Sent message: Message 2 for order order-1
Sent message: Message 3 for order order-1

C:\Presentations\UnlockThePowerOfMessagingPatterns\EventMaterials\CodeMash2025\demos\resilience-and-reliability\sequential-convoy\Producer\bin\Debug\net8.0\Producer.exe (process 12240) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .|
```

```
C:\Presentations\UnlockThePowerOfMessagingPatterns\EventMaterials\CodeMash2025\demos\resilience-and-reliability\sequential-convoy\Consumer\bin\Debug\net8.0\Consumer.exe (process 12244) started.
Waiting for messages...

Accepted session: order-1
Received message: Message 1 for order order-1
Received message: Message 2 for order order-1
Received message: Message 3 for order order-1

Accepted session: order-2
Received message: Message 1 for order order-2
Received message: Message 2 for order order-2
Received message: Message 3 for order order-2
```

# Key Points to Remember

Use Order Keys

Ensure Queue Ordering

Monitor and Adjust

Handle Failures Gracefully

# Wrap-Up

Resilience and Reliability

Unlock the Power of Messaging Patterns

# Key Takeaways

Circuit Breaker

Saga

Sequential Convoy

# Preview: Streaming and Integration Patterns

Event Streaming

Transactional Outbox

# Streaming and Integration Patterns

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Event Streaming

Streaming and Integration Patterns

Unlock the Power of Messaging Patterns

# What is Event Streaming

Continuous flow of data recorded and processed in real time

Enables real-time data processing

# Key Components & Flow



# Key Components & Flow



# Benefits

Low Latency

Scalability

Flexibility

# Drawbacks

Complexity

Infrastructure Setup

Data Processing

Skills Requirements

# Drawbacks

Complexity

Scalability

Resource Intensive

Scalability Challenges

Latency Management

# Drawbacks

Complexity

Scalability

Data Consistency

Resource Intensive

Scalability Challenges

Latency Management

# Use Cases

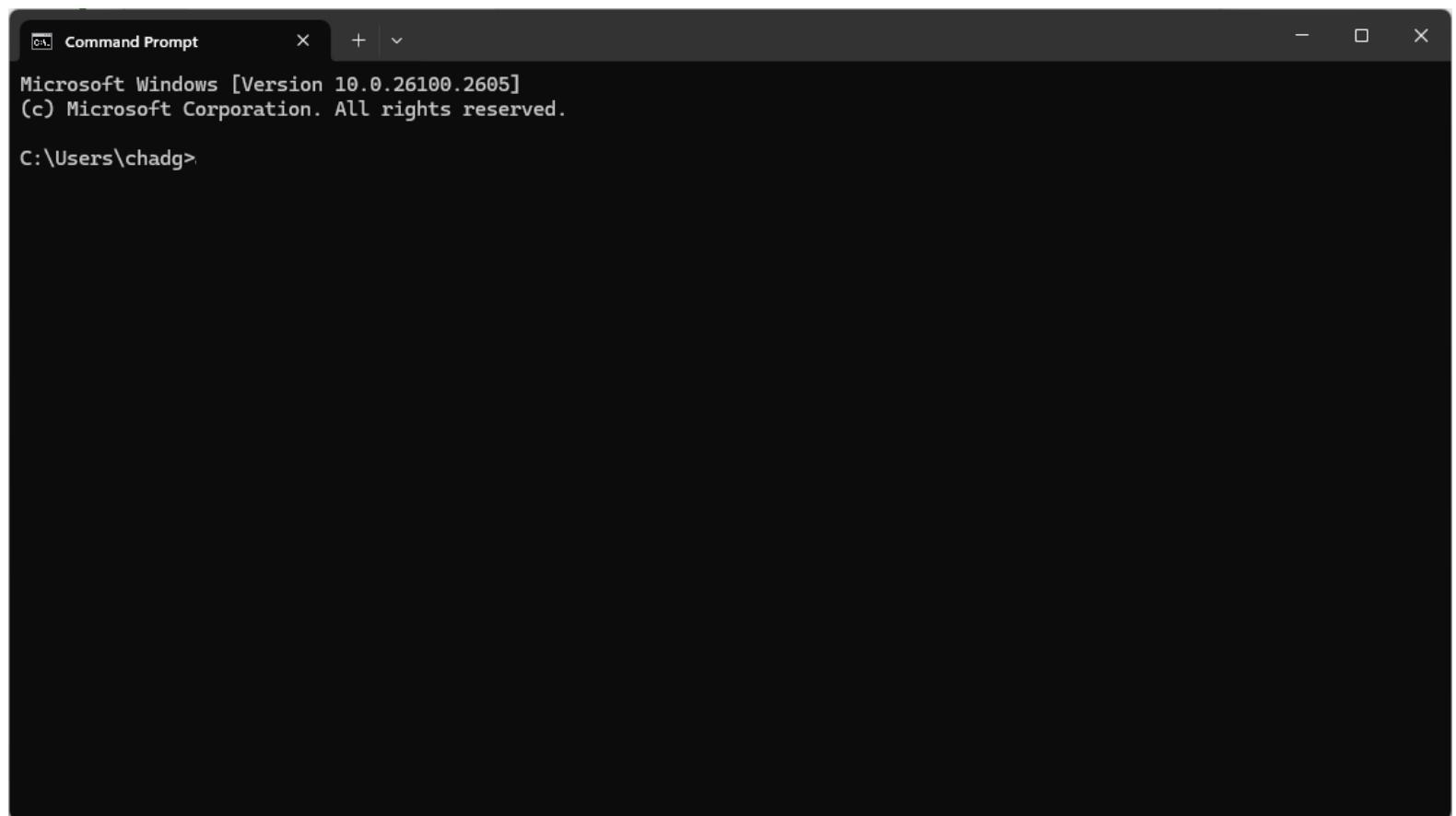
Real-Time  
Analytics

Data Integration

IoT Applications

# Demonstration

Create Namespace

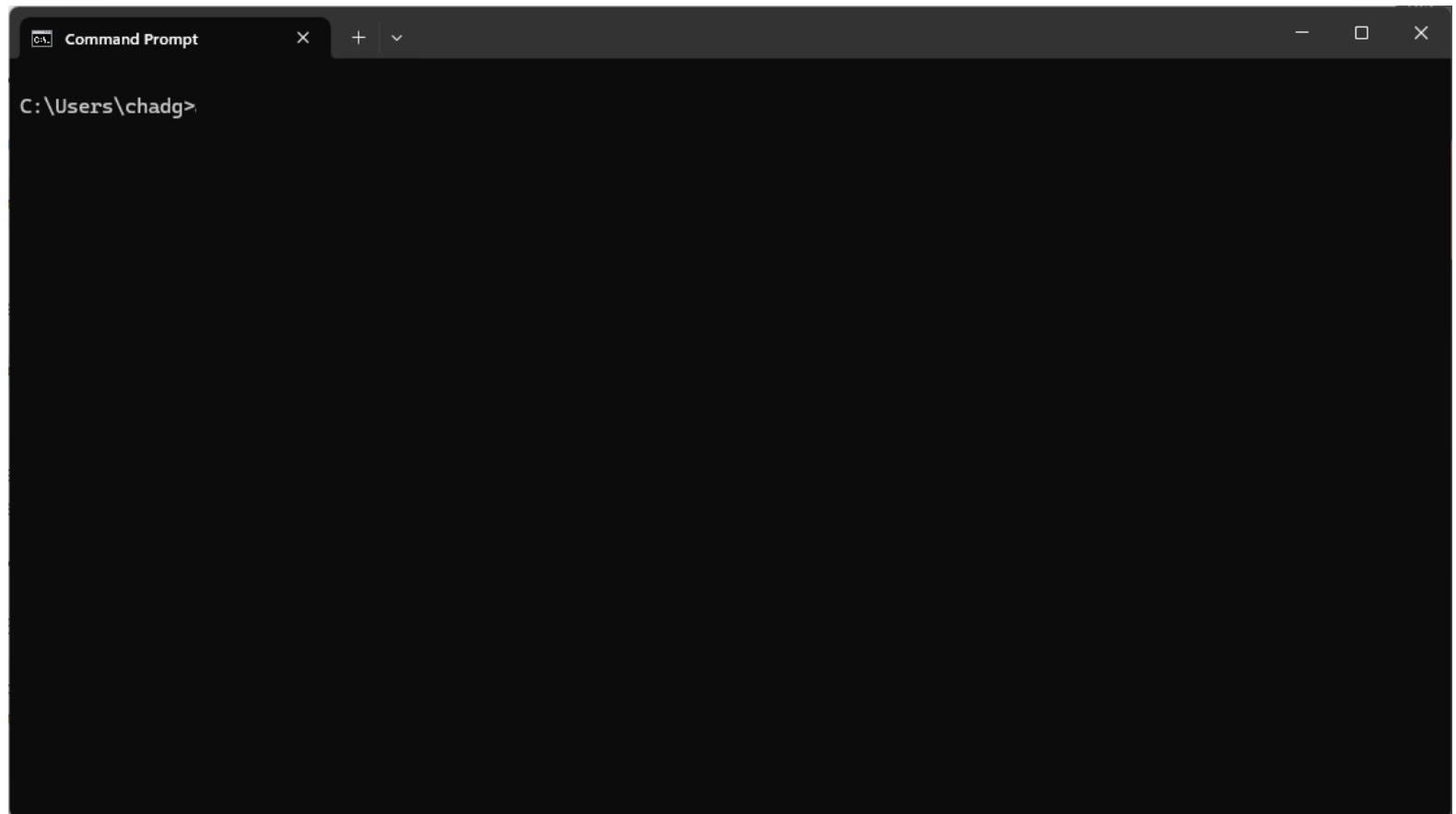


```
Command Prompt
Microsoft Windows [Version 10.0.26100.2605]
(c) Microsoft Corporation. All rights reserved.

C:\Users\chadg>
```

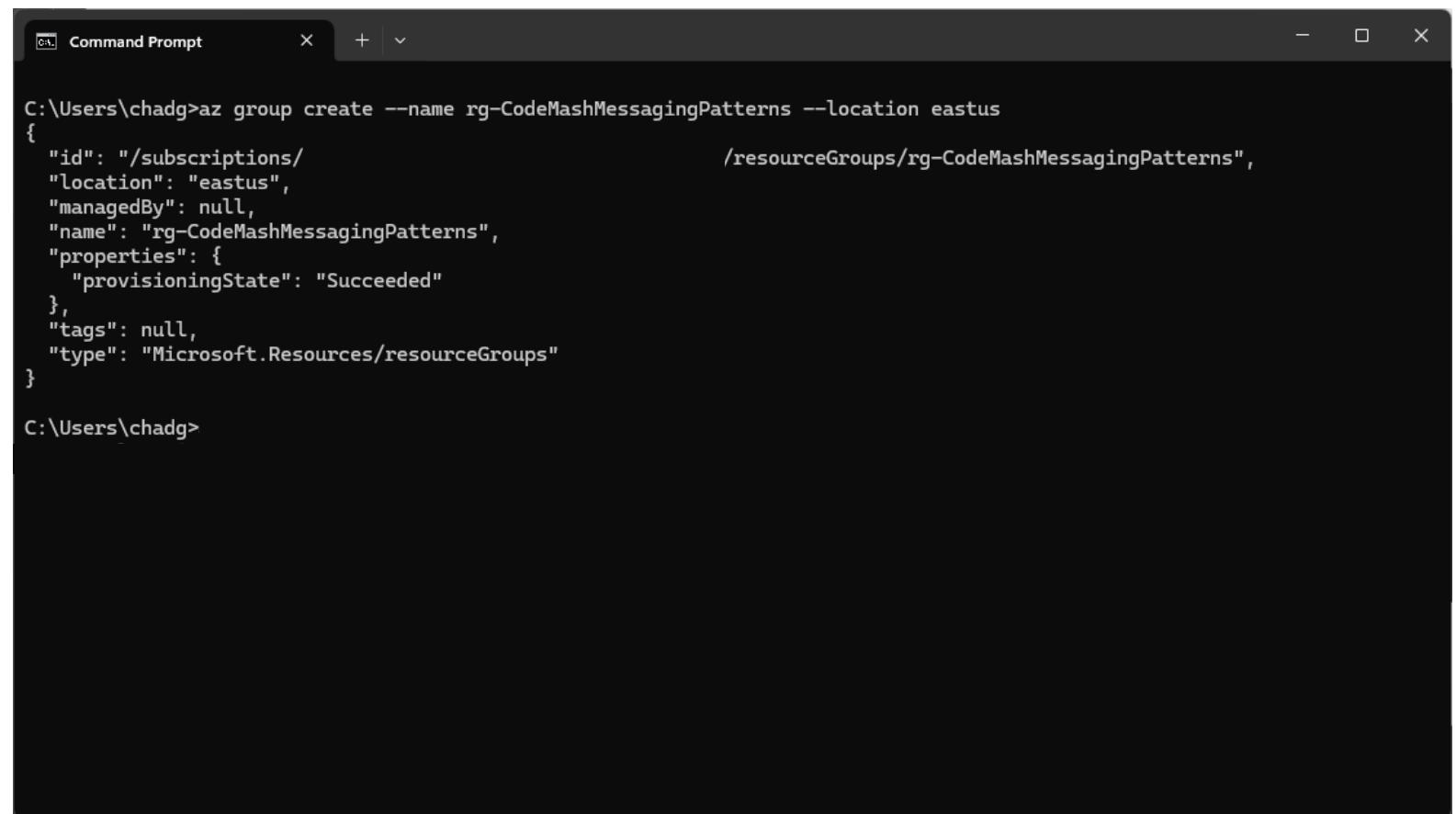
# Demonstration

Create Namespace



# Demonstration

## Create Namespace



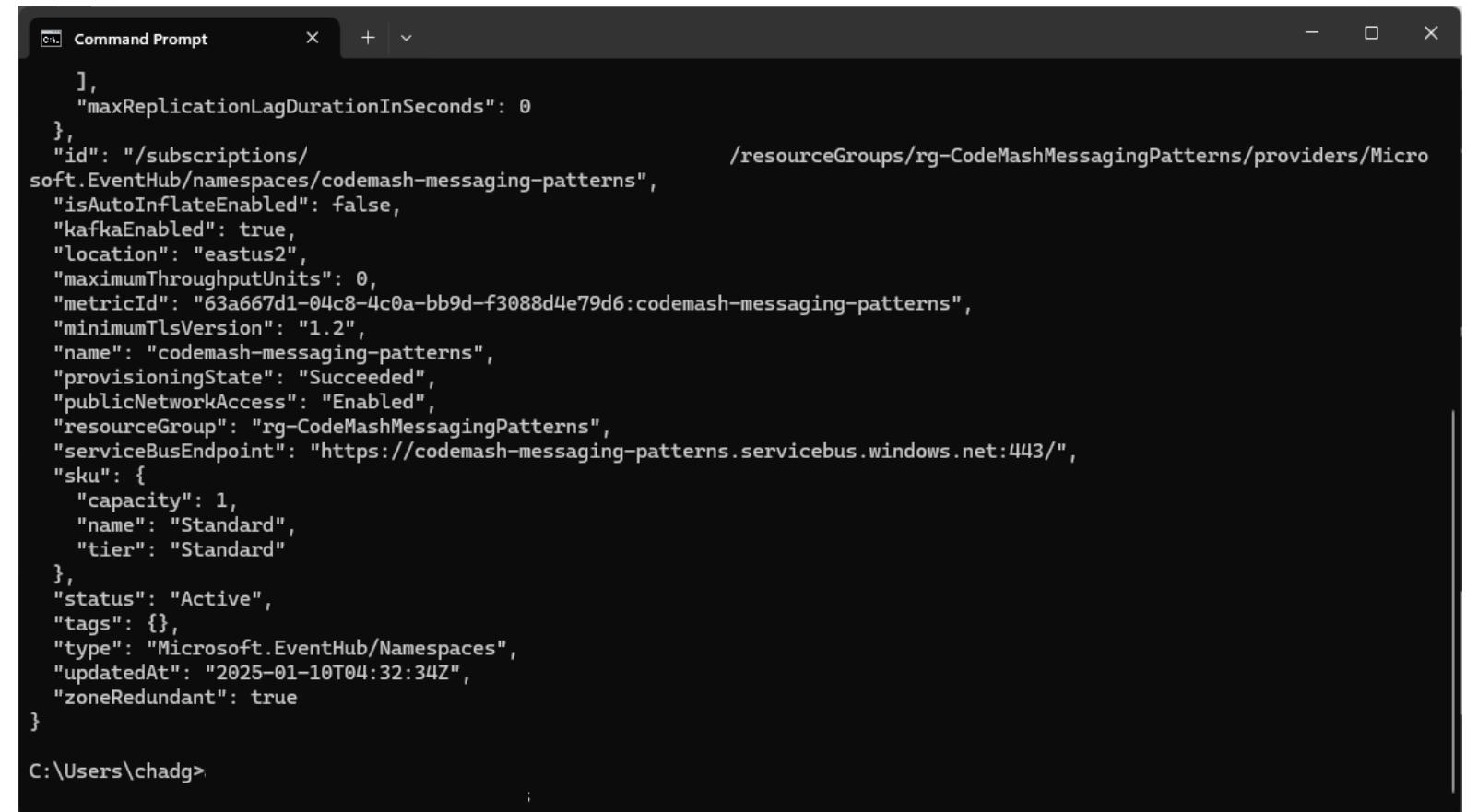
```
C:\Users\chadg>az group create --name rg-CodeMashMessagingPatterns --location eastus
{
  "id": "/subscriptions/",
  "location": "eastus",
  "managedBy": null,
  "name": "rg-CodeMashMessagingPatterns",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}

C:\Users\chadg>
```

# Demonstration

Create Namespace

Create an Event Hub



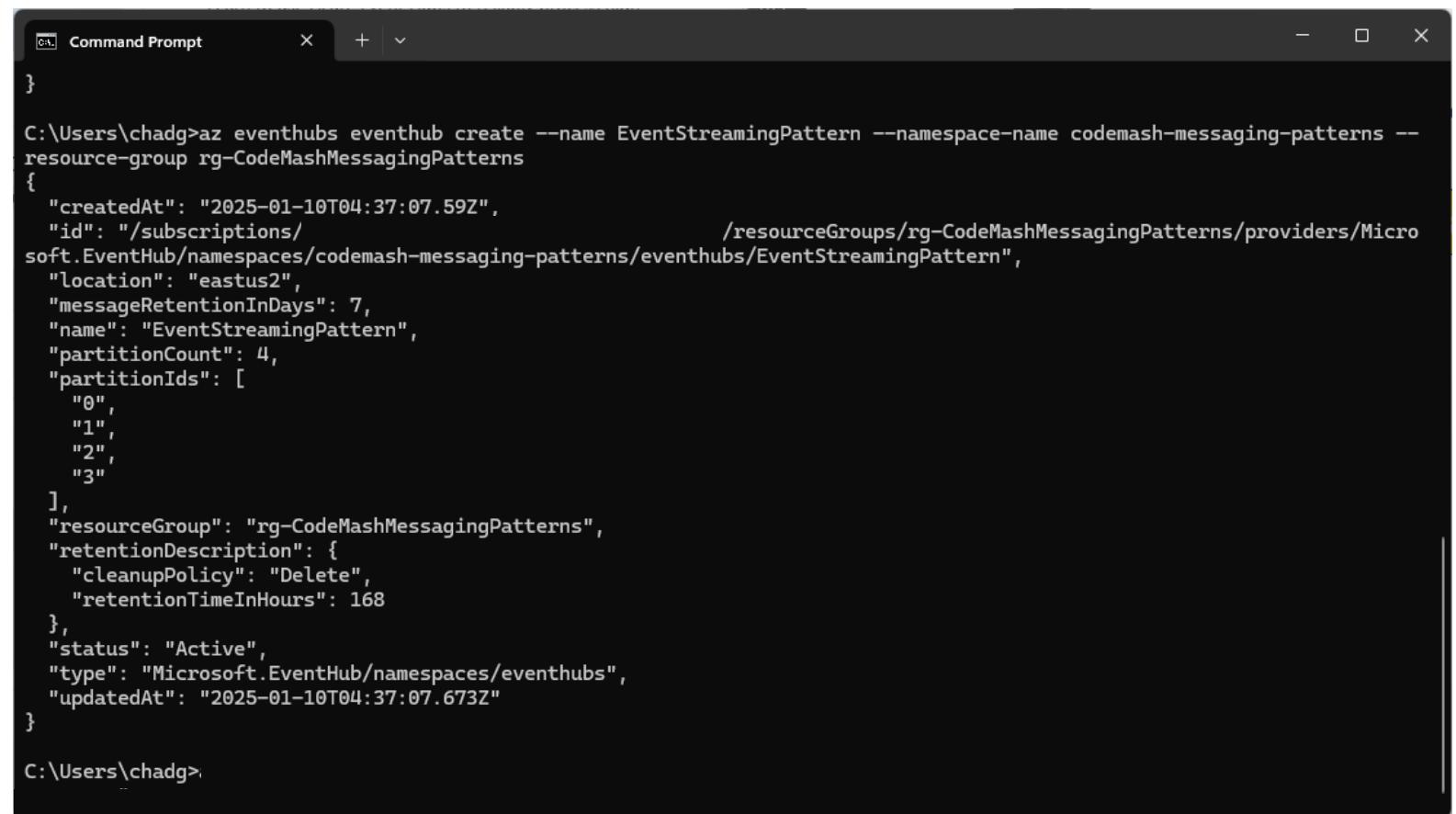
```
Command Prompt      x + v
[{"id": "/subscriptions/soft.EventHub/namespaces/codemash-messaging-patterns", "isAutoInflateEnabled": false, "kafkaEnabled": true, "location": "eastus2", "maximumThroughputUnits": 0, "metricId": "63a667d1-04c8-4c0a-bb9d-f3088d4e79d6:codemash-messaging-patterns", "minimumTlsVersion": "1.2", "name": "codemash-messaging-patterns", "provisioningState": "Succeeded", "publicNetworkAccess": "Enabled", "resourceGroup": "rg-CodeMashMessagingPatterns", "serviceBusEndpoint": "https://codemash-messaging-patterns.servicebus.windows.net:443/", "sku": {"capacity": 1, "name": "Standard", "tier": "Standard"}, "status": "Active", "tags": {}, "type": "Microsoft.EventHub/Namespace", "updatedAt": "2025-01-10T04:32:34Z", "zoneRedundant": true}, /resourceGroups/rg-CodeMashMessagingPatterns/providers/Micro
C:\Users\chadg>
```

# Demonstration

Create Namespace

Create an Event Hub

Assign Roles



```
Command Prompt x + v
}

C:\Users\chadg>az eventhubs eventhub create --name EventStreamingPattern --namespace-name codemash-messaging-patterns --resource-group rg-CodeMashMessagingPatterns
{
  "createdAt": "2025-01-10T04:37:07.59Z",
  "id": "/subscriptions/ /resourceGroups/rg-CodeMashMessagingPatterns/providers/Microsoft.EventHub/namespaces/codemash-messaging-patterns/eventhubs/EventStreamingPattern",
  "location": "eastus2",
  "messageRetentionInDays": 7,
  "name": "EventStreamingPattern",
  "partitionCount": 4,
  "partitionIds": [
    "0",
    "1",
    "2",
    "3"
  ],
  "resourceGroup": "rg-CodeMashMessagingPatterns",
  "retentionDescription": {
    "cleanupPolicy": "Delete",
    "retentionTimeInHours": 168
  },
  "status": "Active",
  "type": "Microsoft.EventHub/namespaces/eventhubs",
  "updatedAt": "2025-01-10T04:37:07.673Z"
}

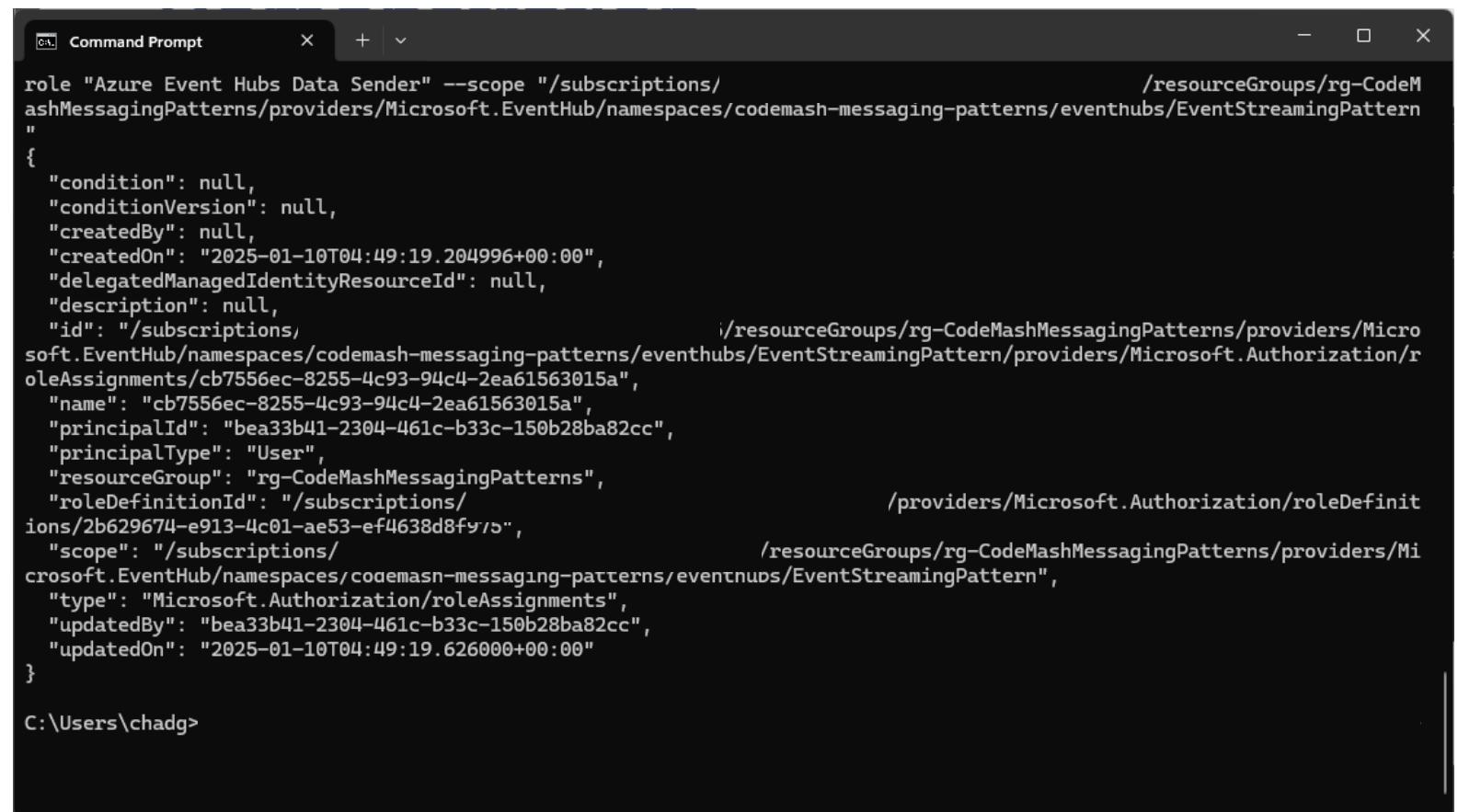
C:\Users\chadg>
```

# Demonstration

Create Namespace

Create an Event Hub

Assign Roles



```
Command Prompt x + v
role "Azure Event Hubs Data Sender" --scope "/subscriptions/
ashMessagingPatterns/providers/Microsoft.EventHub/namespaces/codemash-messaging-patterns/eventhubs/EventStreamingPattern
"
{
  "condition": null,
  "conditionVersion": null,
  "createdBy": null,
  "createdOn": "2025-01-10T04:49:19.204996+00:00",
  "delegatedManagedIdentityResourceId": null,
  "description": null,
  "id": "/subscriptions/
                                /resourceGroups/rg-CodeM
soft.EventHub/namespaces/codemash-messaging-patterns/eventhubs/EventStreamingPattern/providers/Microsoft.Authorization/r
oleAssignments/cb7556ec-8255-4c93-94c4-2ea61563015a",
  "name": "cb7556ec-8255-4c93-94c4-2ea61563015a",
  "principalId": "bea33b41-2304-461c-b33c-150b28ba82cc",
  "principalType": "User",
  "resourceGroup": "rg-CodeMashMessagingPatterns",
  "roleDefinitionId": "/subscriptions/
                                /providers/Microsoft.Authorization/roleDefinit
ions/2b629674-e913-4c01-ae53-ef4638d8fy/s",
  "scope": "/subscriptions/
                                /resourceGroups/rg-CodeMashMessagingPatterns/providers/Mi
crosoft.EventHub/namespaces/codemash-messaging-patterns/eventhubs/EventStreamingPattern",
  "type": "Microsoft.Authorization/roleAssignments",
  "updatedBy": "bea33b41-2304-461c-b33c-150b28ba82cc",
  "updatedOn": "2025-01-10T04:49:19.626000+00:00"
}
C:\Users\chadg>
```

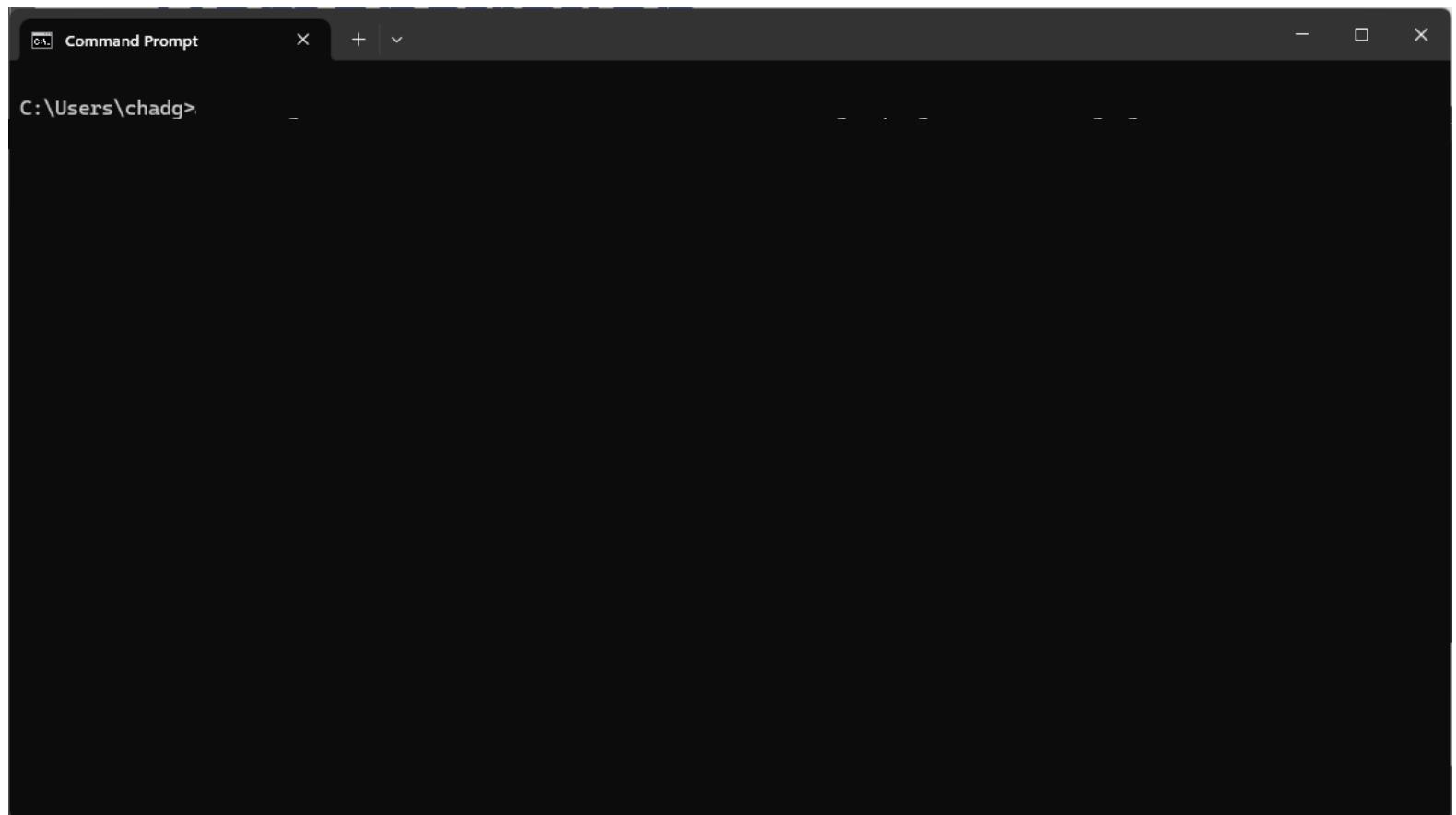
# Demonstration

Create Namespace

Create an Event Hub

Assign Roles

Create Storage Account



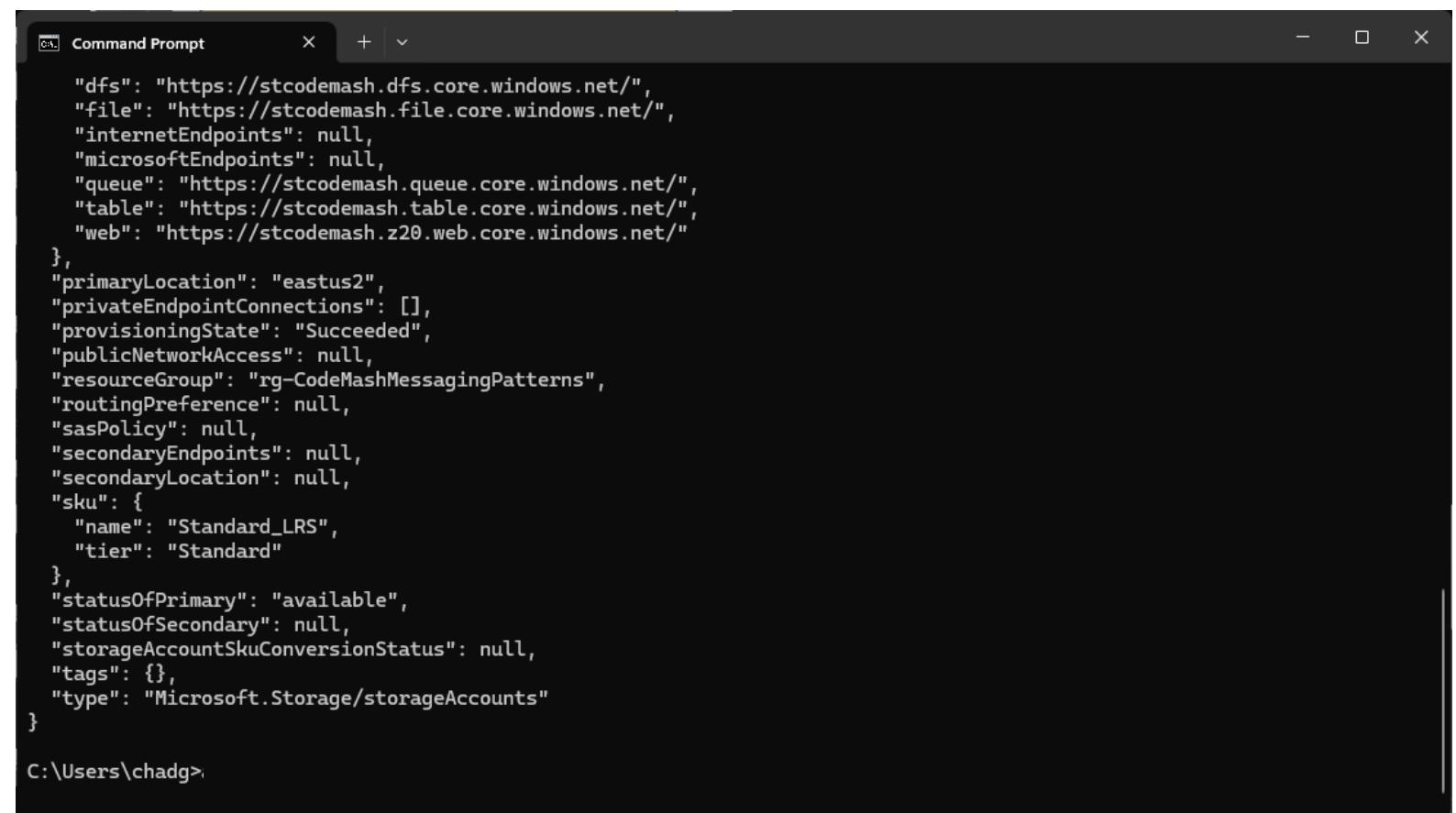
# Demonstration

Create Namespace

Create an Event Hub

Assign Roles

Create Storage Account



```
Command Prompt x + v
{
  "dfs": "https://stcodemash.dfs.core.windows.net/",
  "file": "https://stcodemash.file.core.windows.net/",
  "internetEndpoints": null,
  "microsoftEndpoints": null,
  "queue": "https://stcodemash.queue.core.windows.net/",
  "table": "https://stcodemash.table.core.windows.net/",
  "web": "https://stcodemash.z20.web.core.windows.net/"
},
"primaryLocation": "eastus2",
"privateEndpointConnections": [],
"provisioningState": "Succeeded",
"publicNetworkAccess": null,
"resourceGroup": "rg-CodeMashMessagingPatterns",
"routingPreference": null,
"sasPolicy": null,
"secondaryEndpoints": null,
"secondaryLocation": null,
"sku": {
  "name": "Standard_LRS",
  "tier": "Standard"
},
"statusOfPrimary": "available",
"statusOfSecondary": null,
"storageAccountSkuConversionStatus": null,
"tags": {},
"type": "Microsoft.Storage/storageAccounts"
}

C:\Users\chadg>
```

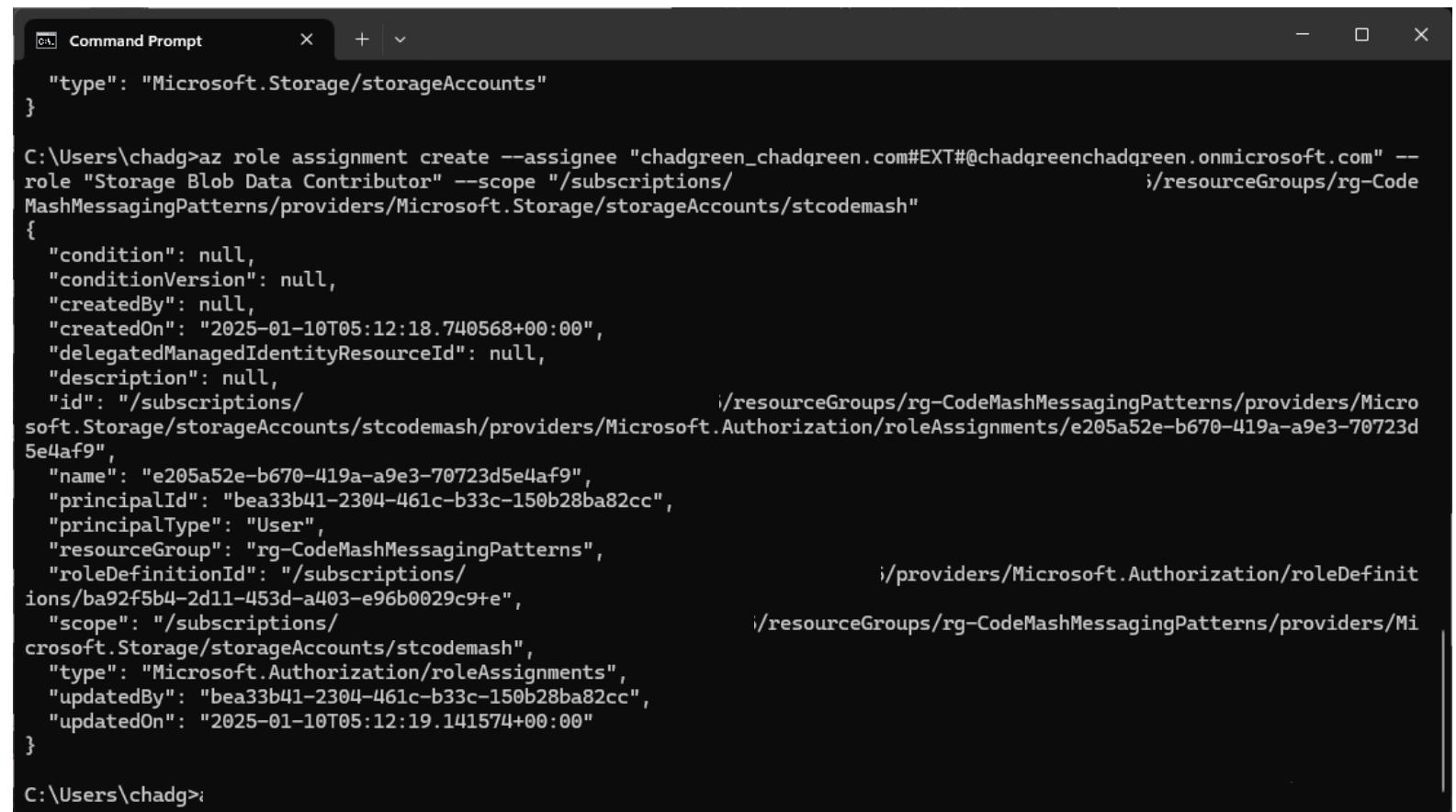
# Demonstration

Create Namespace

Create an Event Hub

Assign Roles

Create Storage Account



```
Command Prompt x + v
"role": "Storage Blob Data Contributor" --scope "/subscriptions/
C:\Users\chadg>az role assignment create --assignee "chadgreen_chadgreen.com#EXT#@chadgreenchadgreen.onmicrosoft.com" --
/resourceGroups/rg-Code
MashMessagingPatterns/providers/Microsoft.Storage/storageAccounts/stcodemash"
{
  "condition": null,
  "conditionVersion": null,
  "createdBy": null,
  "createdOn": "2025-01-10T05:12:18.740568+00:00",
  "delegatedManagedIdentityResourceId": null,
  "description": null,
  "id": "/subscriptions/
/resourceGroups/rg-CodeMashMessagingPatterns/providers/Micro
soft.Storage/storageAccounts/stcodemash/providers/Microsoft.Authorization/roleAssignments/e205a52e-b670-a9e3-70723d
5e4af9",
  "name": "e205a52e-b670-a9e3-70723d5e4af9",
  "principalId": "bea33b41-2304-461c-b33c-150b28ba82cc",
  "principalType": "User",
  "resourceGroup": "rg-CodeMashMessagingPatterns",
  "roleDefinitionId": "/subscriptions/
ions/ba92f5b4-2d11-453d-a403-e96b0029c9+e",
  "scope": "/subscriptions/
crosoft.Storage/storageAccounts/stcodemash",
  "type": "Microsoft.Authorization/roleAssignments",
  "updatedBy": "bea33b41-2304-461c-b33c-150b28ba82cc",
  "updatedOn": "2025-01-10T05:12:19.141574+00:00"
}

C:\Users\chadg>
```

# Demonstration

Create Namespace

```
using Azure.Identity;
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Producer;
using System.Text;
```

Create an Event Hub

```
const string hostName = "codemash-messaging-patterns.servicebus.windows.net";
const string eventHubName = "eventstreamingpattern";
```

Assign Roles

```
// Wait for the user to press a key before sending events
Console.WriteLine("Press any key to send events to the Event Hub...");
Console.ReadKey(true);
```

Create Storage Account

```
// Create a producer client that we can use to send messages to the event hub
await using EventHubProducerClient producer = new(hostName, eventHubName, new DefaultAzureCredential());
```

```
// Create a batch of events
Console.WriteLine();
Console.WriteLine("Creating a batch of events to send to the Event Hub...");
using EventDataBatch eventBatch = await producer.CreateBatchAsync();
for (int i = 0; i < 10; i++)
{
    string eventData = $"Event {i}: {DateTime.UtcNow}";
    eventBatch.TryAdd(new EventData(Encoding.UTF8.GetBytes(eventData)));
    Console.WriteLine($"\\tAdded: {eventData}");
}
```

```
// Send the batch of events to the Event Hub
Console.WriteLine();
Console.WriteLine("Sending the batch of events to the Event Hub...");
await producer.SendAsync(eventBatch);
Console.WriteLine("The batch of events has been sent to the Event Hub.");
```

Create a Message Producer

# Demonstration

Create Namespace

Create an Event Hub

Assign Roles

Create Storage Account

Create a Message Producer

Create a Message Consumer

```
using Azure.Identity;
using Azure.Messaging.EventHubs;
using Azure.Messaging.EventHubs.Consumer;
using Azure.Storage.Blobs;

const string hostName = "codemash-messaging-patterns.servicebus.windows.net";
const string eventHubName = "eventstreamingpattern";
Uri blobContainerUri = new("https://stcodemash.blob.core.windows.net/event-streaming");

BlobContainerClient storageClient = new(
    blobContainerUri, new DefaultAzureCredential());

EventProcessorClient processor = new(
    storageClient,
    EventHubConsumerClient.DefaultConsumerGroupName,
    hostName,
    eventHubName,
    new DefaultAzureCredential());

// Register handlers for processing events and errors
processor.ProcessEventAsync += async (args) =>
{
    Console.WriteLine($"Event received: {args.Data.EventBody}");
    await args.UpdateCheckpointAsync();
};

processor.ProcessErrorAsync += async (args) =>
{
    Console.WriteLine($"Error: {args.Exception.Message}");
    await Task.CompletedTask;
};

await processor.StartProcessingAsync();
Console.WriteLine("Press any key to stop the processor...");
Console.ReadKey(true);

await processor.StopProcessingAsync();
Console.WriteLine("The processor has been stopped.");
```

Unlock the Power of Messaging Patterns

# Demonstration

Create Namespace

Create an Event Hub

Assign Roles

Create Storage Account

Create a Message Producer

Create a Message Consumer

Run the Demo

The screenshot shows a terminal window titled "Producer". It displays a series of events being added to a batch, followed by a message indicating the batch has been sent to the Event Hub.

```
Press any key to send events to the Event Hub...
Creating a batch of events to send to the Event Hub...
Added: Event 0: 1/10/2025 9:32:09 PM
Added: Event 1: 1/10/2025 9:32:09 PM
Added: Event 2: 1/10/2025 9:32:09 PM
Added: Event 3: 1/10/2025 9:32:09 PM
Added: Event 4: 1/10/2025 9:32:09 PM
Added: Event 5: 1/10/2025 9:32:09 PM
Added: Event 6: 1/10/2025 9:32:09 PM
Added: Event 7: 1/10/2025 9:32:09 PM
Added: Event 8: 1/10/2025 9:32:09 PM
Added: Event 9: 1/10/2025 9:32:09 PM

Sending the batch of events to the Event Hub...
The batch of events has been sent to the Event Hub.

C:\Presentations\UnlockThePowerOfMessagingPatterns\EventMaterials\CodeMash2025\demos\advanced-messaging-patterns\event-streaming\Producer\bin\Debug\net8.0\Producer.exe (process 27912) exited with code 0 (0x0).
```

The screenshot shows a terminal window titled "Consumer". It displays a series of events received from the Event Hub, with a message prompting the user to stop the processor.

```
Press any key to stop the processor...
Event received: Event 0: 1/10/2025 9:32:09 PM
Event received: Event 1: 1/10/2025 9:32:09 PM
Event received: Event 2: 1/10/2025 9:32:09 PM
Event received: Event 3: 1/10/2025 9:32:09 PM
Event received: Event 4: 1/10/2025 9:32:09 PM
Event received: Event 5: 1/10/2025 9:32:09 PM
Event received: Event 6: 1/10/2025 9:32:09 PM
Event received: Event 7: 1/10/2025 9:32:09 PM
Event received: Event 8: 1/10/2025 9:32:09 PM
Event received: Event 9: 1/10/2025 9:32:09 PM
```

# Key Points to Remember

- Real-time processing on data enables quick decision-making and responsiveness.
- Event streaming supports high volumes of data and multiple consumers.
- Enhances data integration and real-time analytics capabilities.

# Wrap-Up

Streaming and Integration Patterns

Unlock the Power of Messaging Patterns

# Key Takeaways

Event Streaming

Transactional  
Outbox

# Preview: Design Considerations

Reliability

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Reliability

Design Considerations

Unlock the Power of Messaging Patterns

# Reliability

Message Delivery and  
Processing

# Reliability

Message Delivery and Processing

Failure Handling

# Reliability

Message Delivery and Processing

Failure Handling

Data Consistency

# Reliability

Message Delivery and Processing

Failure Handling

Data Consistency

# Ensuring Message Delivery and Processing



# Ensuring Message Delivery and Processing



# Ensuring Message Delivery and Processing

Guaranteed Delivery

# Ensuring Message Delivery and Processing

Guaranteed Delivery

Message Persistence

# Acknowledgement Protocols

Guaranteed Delivery

## Positive Acknowledgements



## Negative Acknowledgements



# Acknowledgement Protocols

Guaranteed Delivery

## Positive Acknowledgements

```
await args.CompleteMessageAsync(args.Message);
```

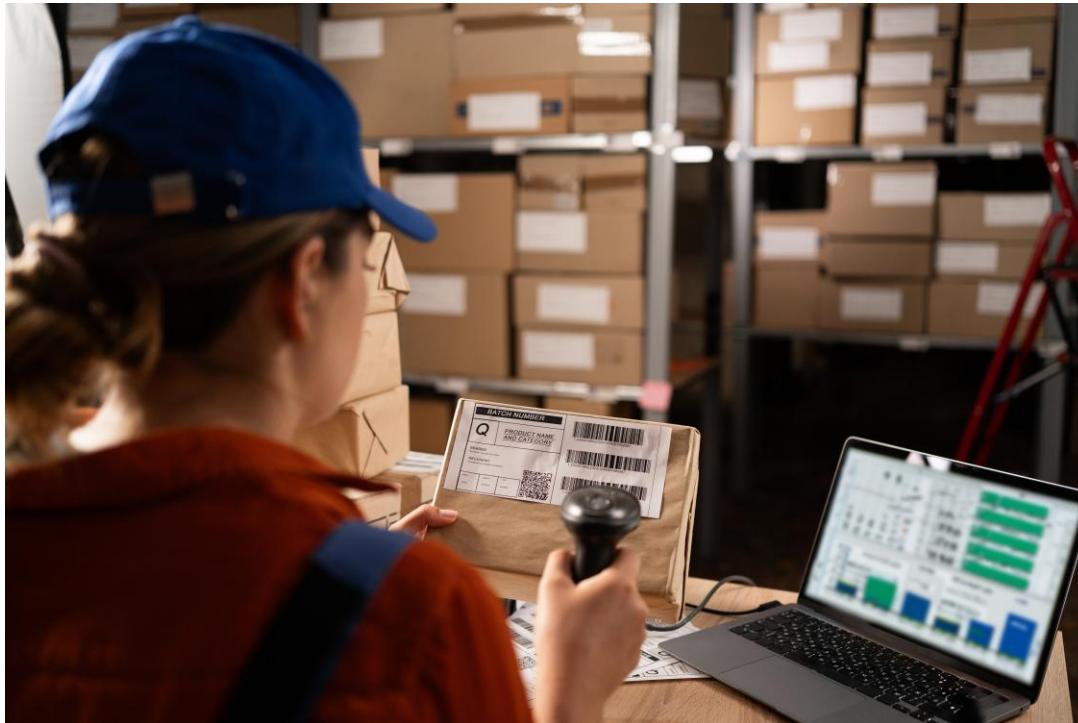
## Negative Acknowledgements

```
await args.AbandonMessageAsync(args.Message);
```

# End-to-End Acknowledgements

Guaranteed Delivery

## Tracking Identifiers



## Delivery Receipts

<b>SENDER: COMPLETE THIS SECTION</b>		<b>COMPLETE THIS SECTION ON DELIVERY</b>							
<ul style="list-style-type: none"><li>■ Complete items 1, 2, and 3. Also complete item 4 if Restricted Delivery is desired.</li><li>■ Print your name and address on the reverse so that we can return the card to you.</li><li>■ Attach this card to the back of the mailpiece, or on the front if space permits.</li></ul>		<p>A. Signature <input checked="" type="checkbox"/> Agent <input type="checkbox"/> Addressee <b>X</b></p> <p>B. Received by (Printed Name) <input type="checkbox"/> C. Date of Delivery</p> <p>D. Is delivery address different from item 1? <input type="checkbox"/> Yes If YES, enter delivery address below: <input type="checkbox"/> No</p> <p>E. Service Type</p> <table border="0"><tr><td><input type="checkbox"/> Certified Mail</td><td><input type="checkbox"/> Express Mail</td></tr><tr><td><input type="checkbox"/> Registered</td><td><input type="checkbox"/> Return Receipt for Merchandise</td></tr><tr><td><input type="checkbox"/> Insured Mail</td><td><input type="checkbox"/> C.O.D.</td></tr></table> <p>F. Restricted Delivery? (Extra Fee) <input type="checkbox"/> Yes</p>		<input type="checkbox"/> Certified Mail	<input type="checkbox"/> Express Mail	<input type="checkbox"/> Registered	<input type="checkbox"/> Return Receipt for Merchandise	<input type="checkbox"/> Insured Mail	<input type="checkbox"/> C.O.D.
<input type="checkbox"/> Certified Mail	<input type="checkbox"/> Express Mail								
<input type="checkbox"/> Registered	<input type="checkbox"/> Return Receipt for Merchandise								
<input type="checkbox"/> Insured Mail	<input type="checkbox"/> C.O.D.								
<b>2. Article Number</b> (Transfer from service label)									
PS Form 3811, February 2004		Domestic Return Receipt							
102595-02-M-1540									

# Ensuring Message Delivery and Processing



Unlock the Power of Messaging Patterns

# Durable Queues and Topics

Message Persistence

Disk-Based Storage



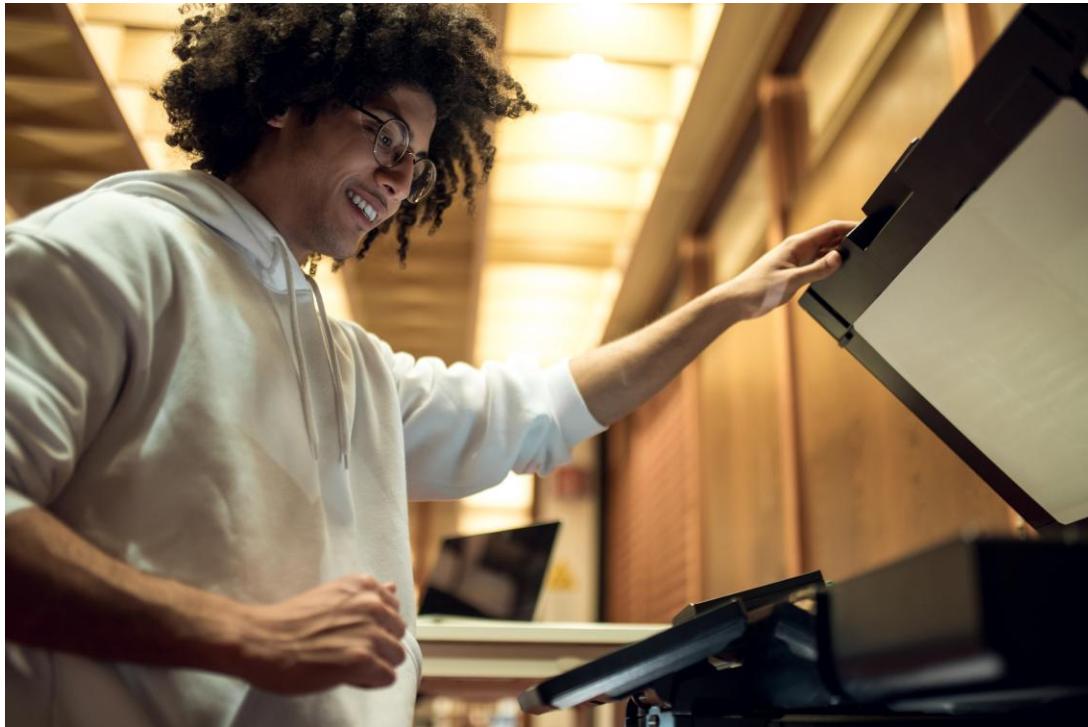
Cloud Storage



Unlock the Power of Messaging Patterns

# Message Replication

Message Persistence



**Synchronous Replication**

**Asynchronous Replication**

# Acknowledgement-Based Persistence

Message Persistence



**Write-Ahead Logging (WAL)**

**Two-Phase Commit (2PC)**

# Redundant Storage Solutions

Message Persistence



RAID

Geo-Replication

# Monitoring and Alerting

Message Persistence

## Health Checks



## Alerts and Notifications



Unlock the Power of Messaging Patterns

# Key Points to Remember

## Message Persistence

- Durable queues and topics ensure messages stored persistently
- Replication provides redundancy and high availability
- Acknowledgement-based persistent ensures messages are safely stored before being acknowledged.
- Redundant storage solutions enhance durability and availability
- Monitoring and alerting help main health and performance

# Key Points to Remember

## Message Persistence

- Durable queues and topics ensure messages are stored persistently
- Replication provides fault tolerance and availability
- Acknowledgements ensure messages are safely stored before being processed
- Redundant storage and failover ensure high availability
- Monitoring and alerts ensure system performance



# Key Points to Remember

Message Persistence



# Reliability

Message Delivery and Processing

Failure Handling

Data Consistency

# Strategies for Handling Failures and Retries



Unlock the Power of Messaging Patterns

# Retry Policies



Unlock the Power of Messaging Patterns

# Exponential Backoff

Retry Policies



Exponential Increase

Cap Maximum Wait Time

# Jitter

## Retry Policies



**Random Jitter**

**Decorrelated Jitter**

# Fixed and Incremental Delays

## Retry Policies



Fixed Delay

Incremental Delay

# Retry Limits

## Retry Policies



Maximum Retry Attempts

Retry Time Window

# Contextual and Conditional Retries

## Retry Policies



Failure Analysis

Conditional Retries

# Dead Letter Queues



**Automatic Routing**

**Monitoring and Analysis**

# Key Points to Remember

## Retry Policies

- **Exponential Backoff and Jitter:** Use to manage retry intervals
- **Fixed and Incremental Delays:** Strategies for consistent retry intervals.
- **Retry Limits:** Set limits to prevent infinite retries.
- **Contextual and Conditional Retries:** Tailor retries based on context and conditions.
- **Dead Letter Queues:** Capture persistent failures.

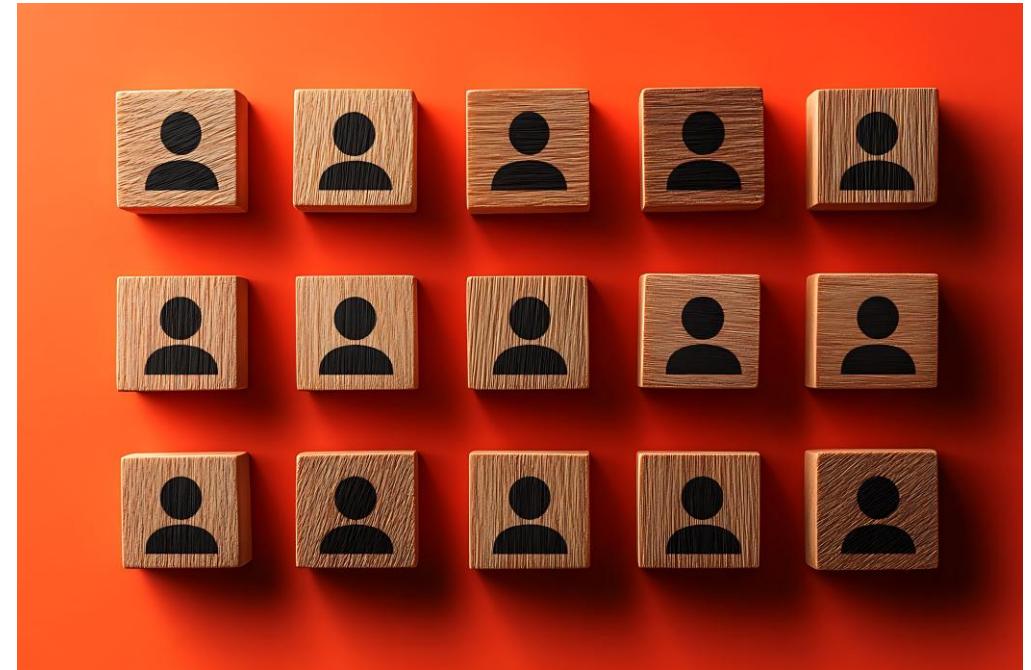
# Reliability

Message Delivery and Processing

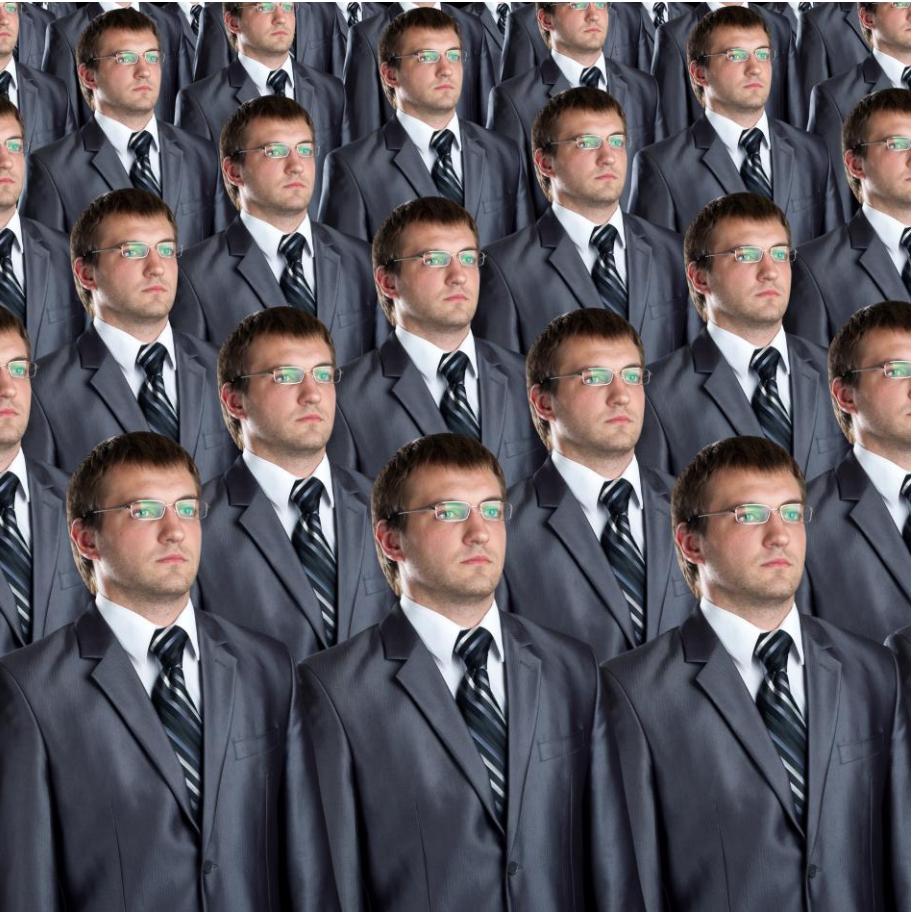
Failure Handling

Data Consistency

# Techniques for Ensuring Data Consistency

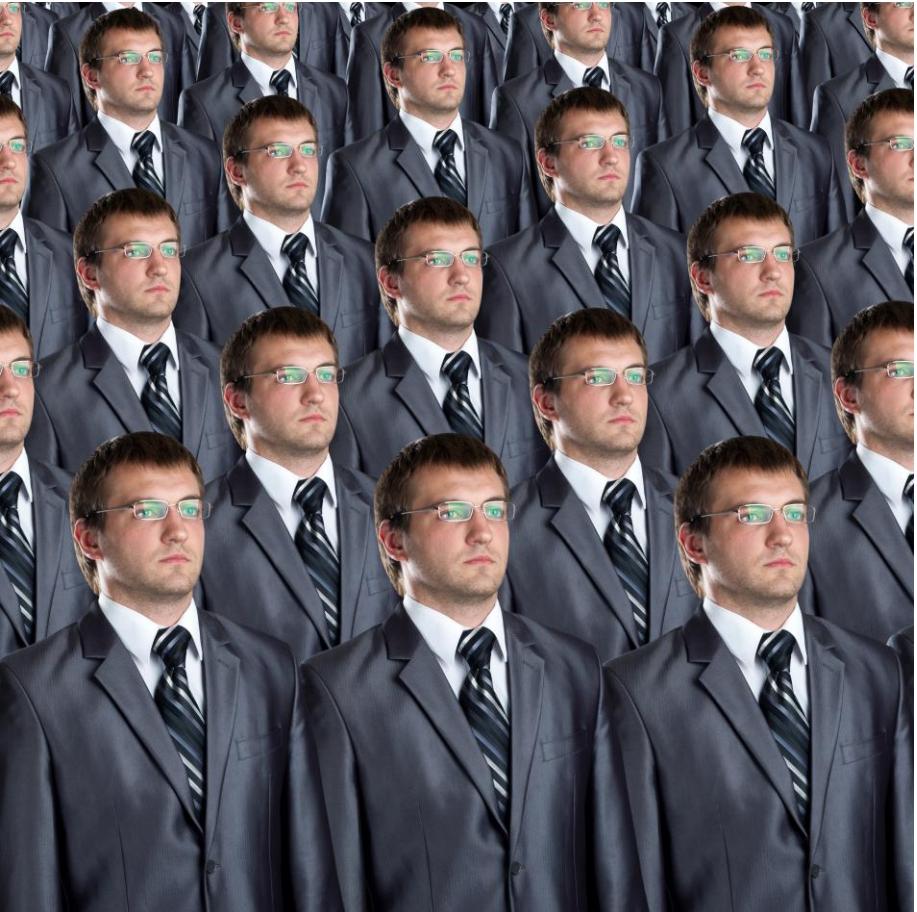


# Idempotency



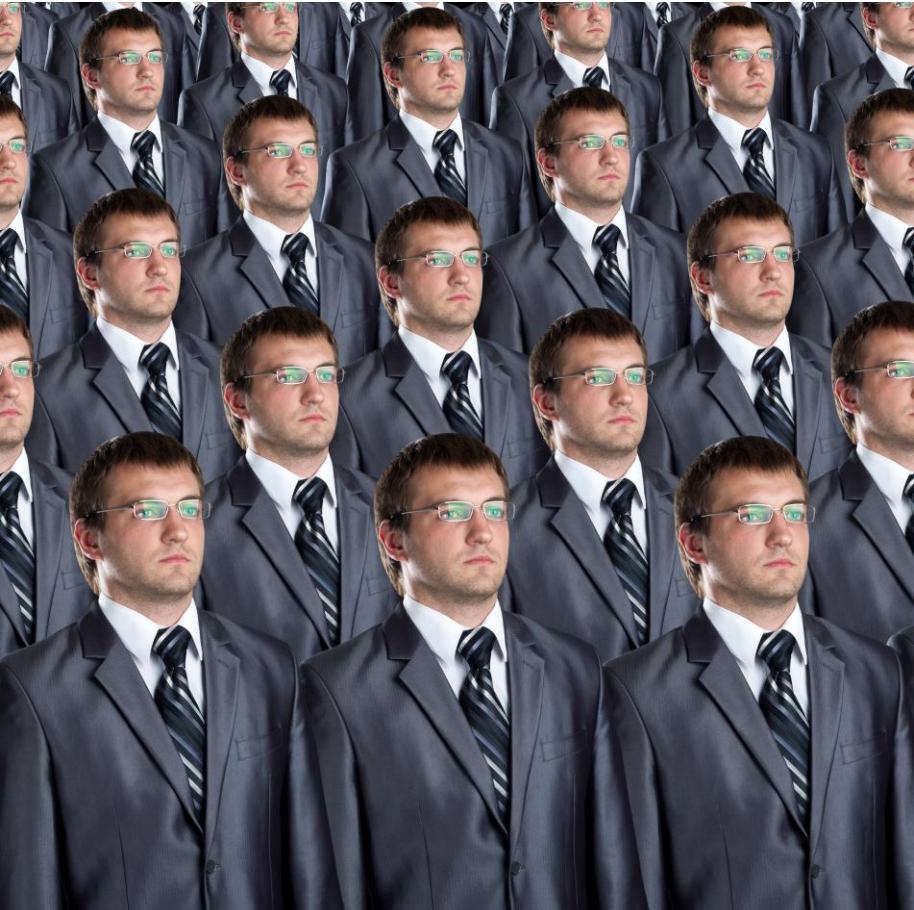
Unlock the Power of Messaging Patterns

# Idempotency



Unique Identifiers

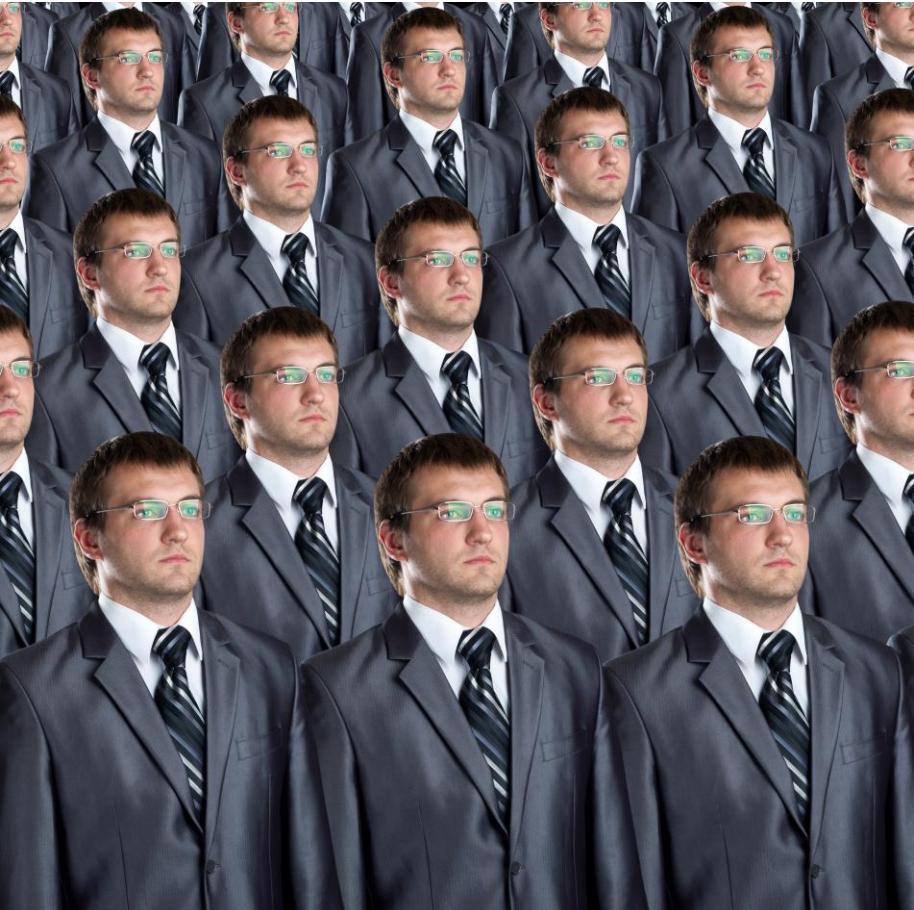
# Idempotency



Unique Identifiers

Idempotent Operations

# Idempotency



Unique Identifiers

Idempotent Operations

State Management

# Transactional Messaging



Unlock the Power of Messaging Patterns

# Transactional Messaging



Unique Identifiers

# Transactional Messaging



Unique Identifiers

Two-Chase Commit

# Transactional Messaging



Unique Identifiers

Two-Chase Commit (2PC)

Compensating Transactions

# Transactional Messaging



Unique Identifiers

Two-Chase Commit (2PC)

Compensating Transactions

Transactional Middleware

# Key Points to Remember

Techniques for Ensuring Data Consistency

- **Idempotency:** Ensure operations can be performed multiple times without changing the result.
- **Transactional Messaging:** Ensure message processing is atomic and consistent.

# Recap: Reliability

Guaranteed Delivery  
and Message  
Persistence

Retry Policies and  
Dead Letter Queues

Idempotency and  
Transactional  
Messaging

# Scalability

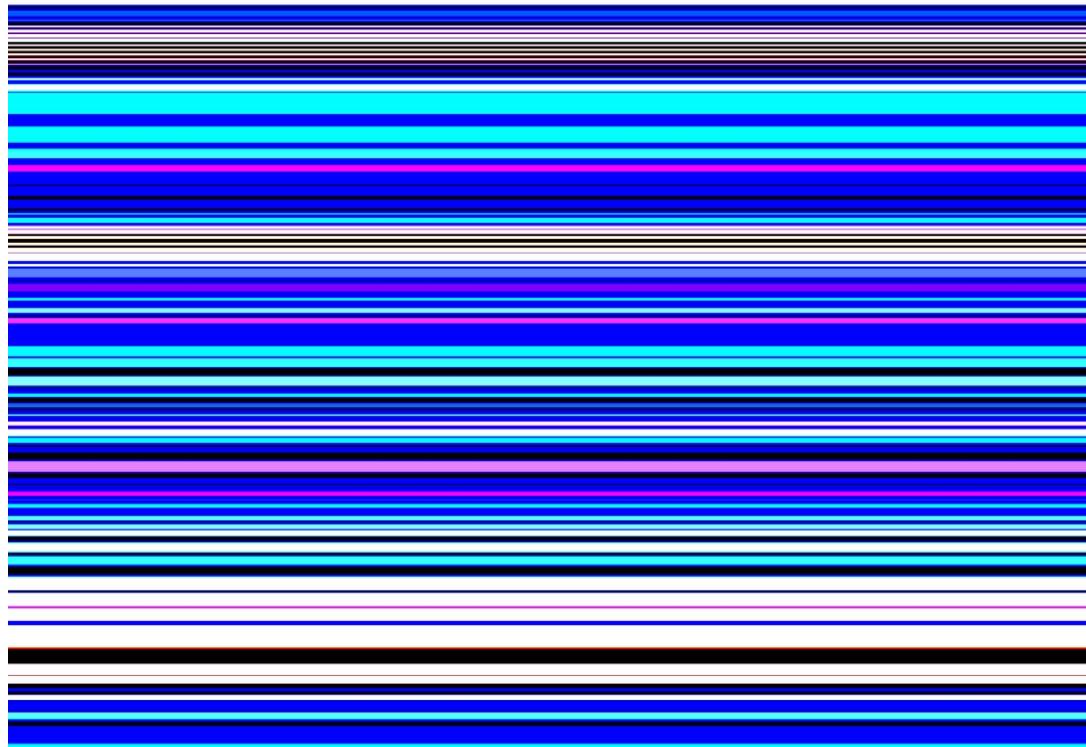
Design Considerations

Unlock the Power of Messaging Patterns

# Designing for Horizontal and Vertical Scaling

# Horizontal Scaling

Designing for Horizontal and Vertical Scaling

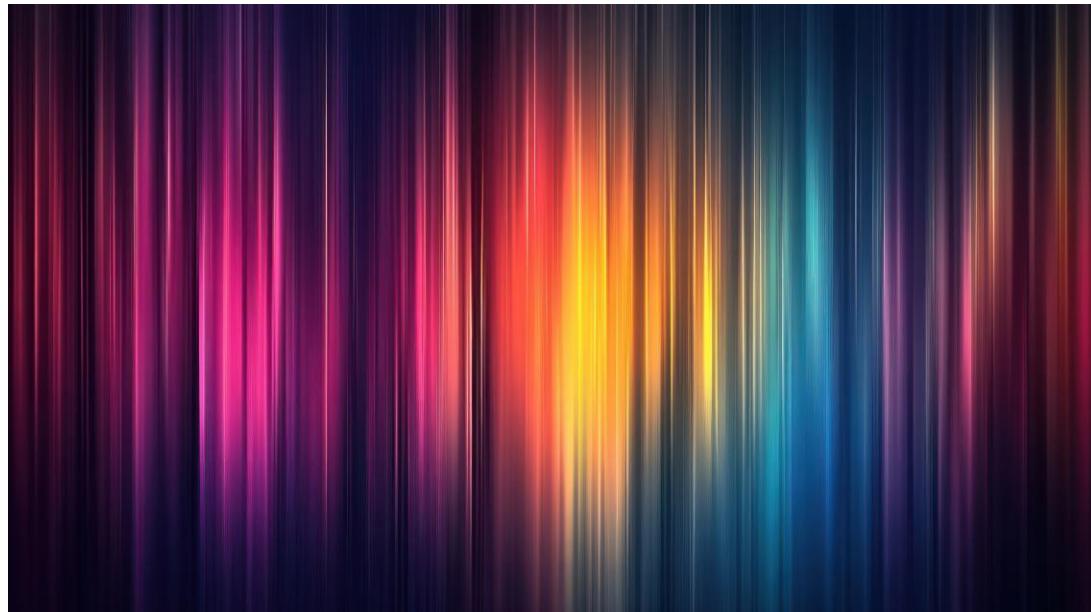


Load Balancers

Sharding

# Vertical Scaling

Designing for Horizontal and Vertical Scaling



Resource Optimization

Scaling Up

# Load Balancing Techniques



Unlock the Power of Messaging Patterns

# Load Balancing Techniques

Round Robin



# Load Balancing Techniques



Round Robin

Weighted Round Robin

# Load Balancing Techniques



Round Robin

Weighted Round Robin

Least Connections

# Load Balancing Techniques



Round Robin

Weighted Round Robin

Least Connections

Content-Based Routing

# Strategies for Handling High-Throughput and Large Volumes of Messages



# Strategies for Handling High-Throughput and Large Volumes of Messages

Batch Processing



# Strategies for Handling High-Throughput and Large Volumes of Messages



Batch Processing

Message Partitioning

# Strategies for Handling High-Throughput and Large Volumes of Messages



Batch Processing

Message Partitioning

Asynchronous Processing

# Strategies for Handling High-Throughput and Large Volumes of Messages



Batch Processing

Message Partitioning

Asynchronous Processing

Auto-Scaling

# Recap: Scalability

Horizontal and  
Vertical Scaling

Load Balancing  
Techniques

High-Throughput  
Strategies

# Performance

Design Considerations

Unlock the Power of Messaging Patterns

# Minimizing Latency and Maximizing Throughput

# Latency

Minimizing Latency and Maximizing Throughput



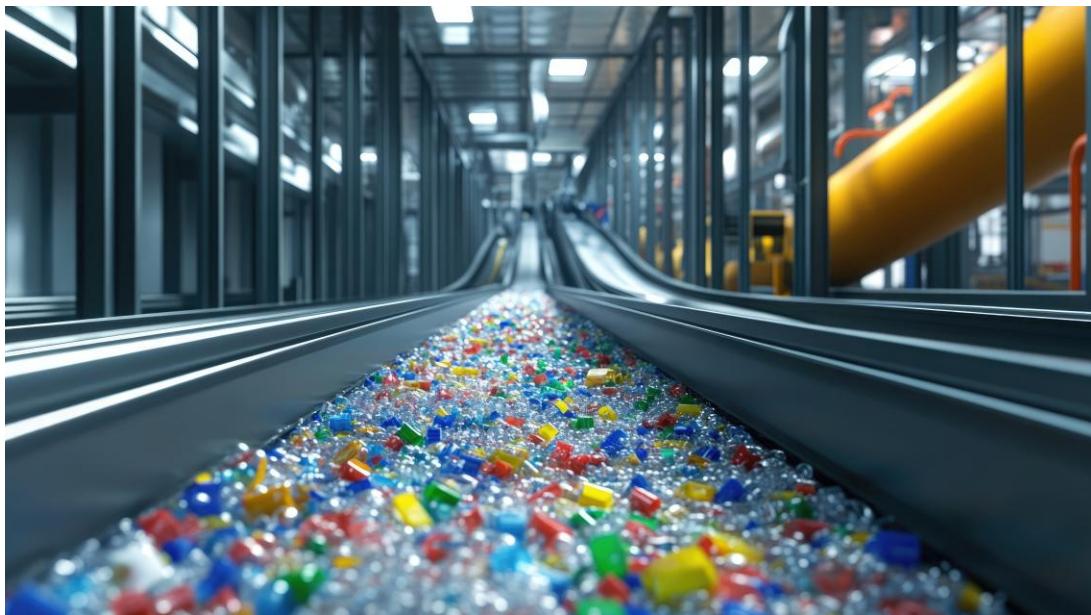
**Efficient Protocols**

**Geographic Proximity**

**Caching**

# Throughput

Minimizing Latency and Maximizing Throughput

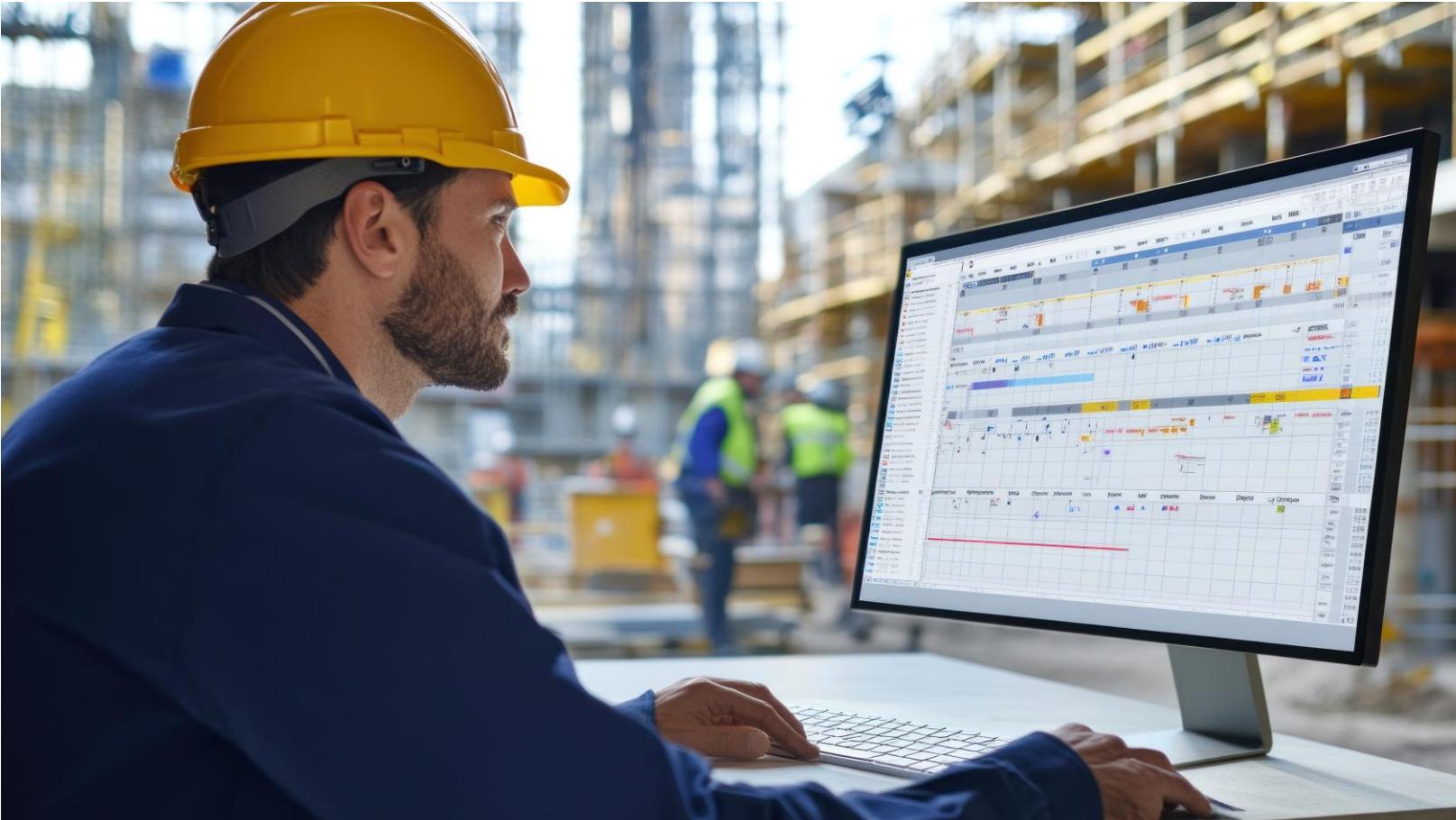


**Batch Processing**

**Asynchronous Processing**

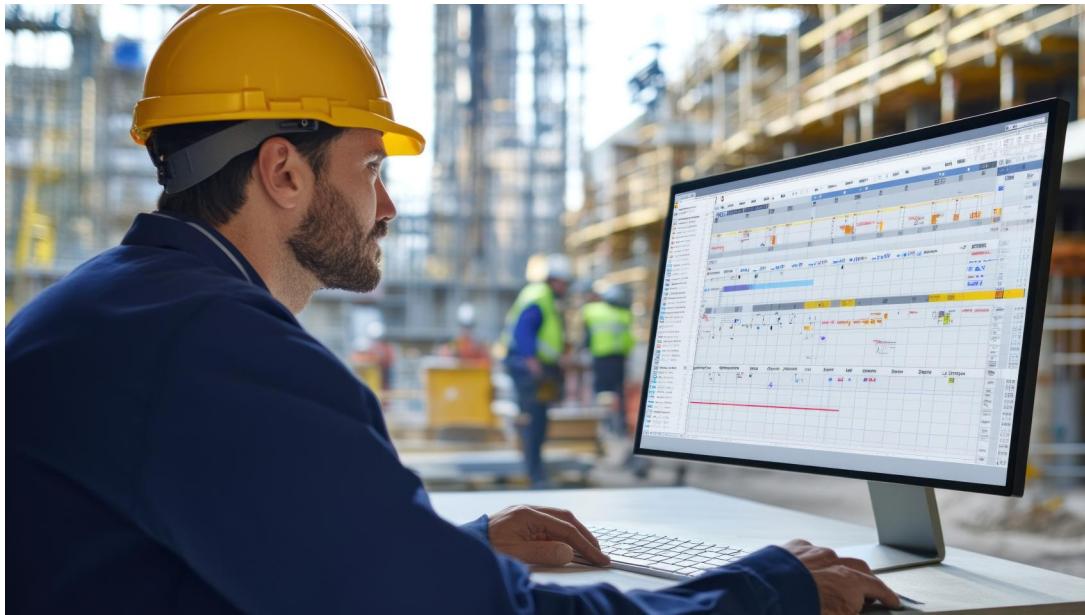
**Load Balancing**

# Optimizing Message Processing



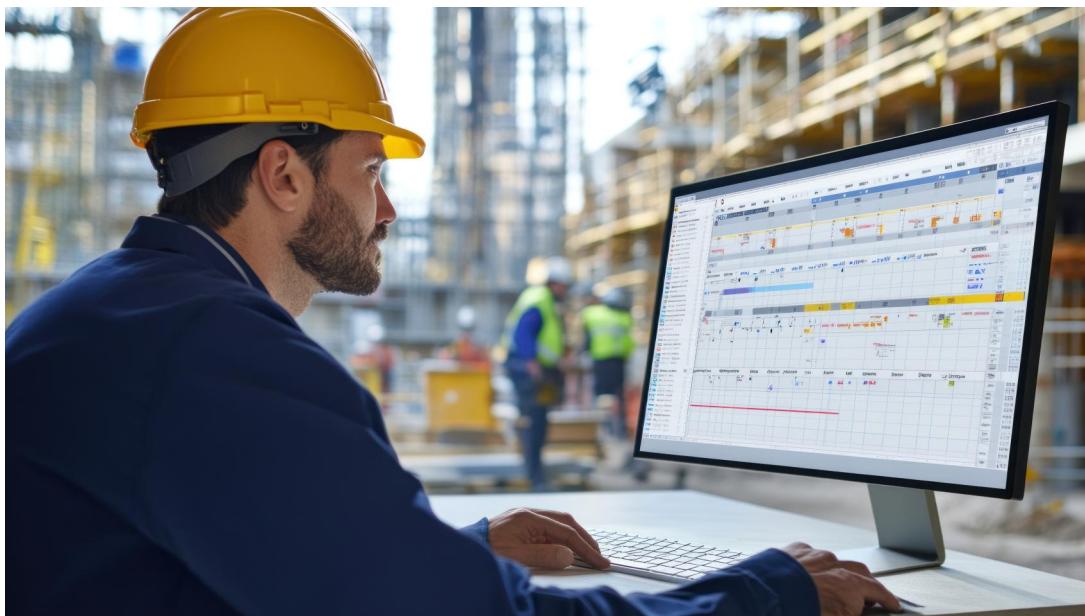
Unlock the Power of Messaging Patterns

# Optimizing Message Processing



Efficient Serialization

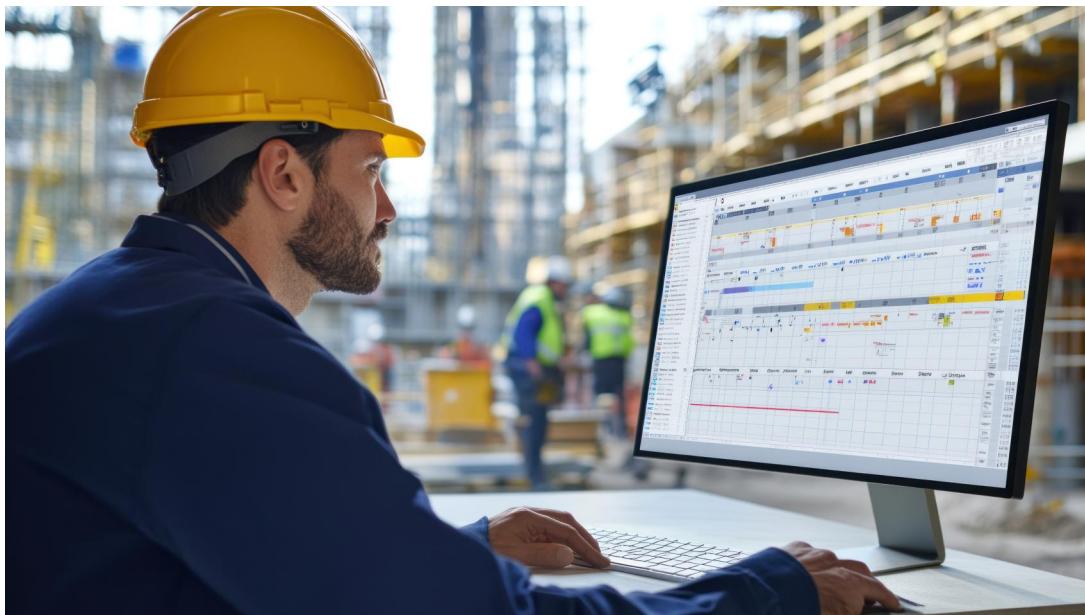
# Optimizing Message Processing



Efficient Serialization

Compression

# Optimizing Message Processing

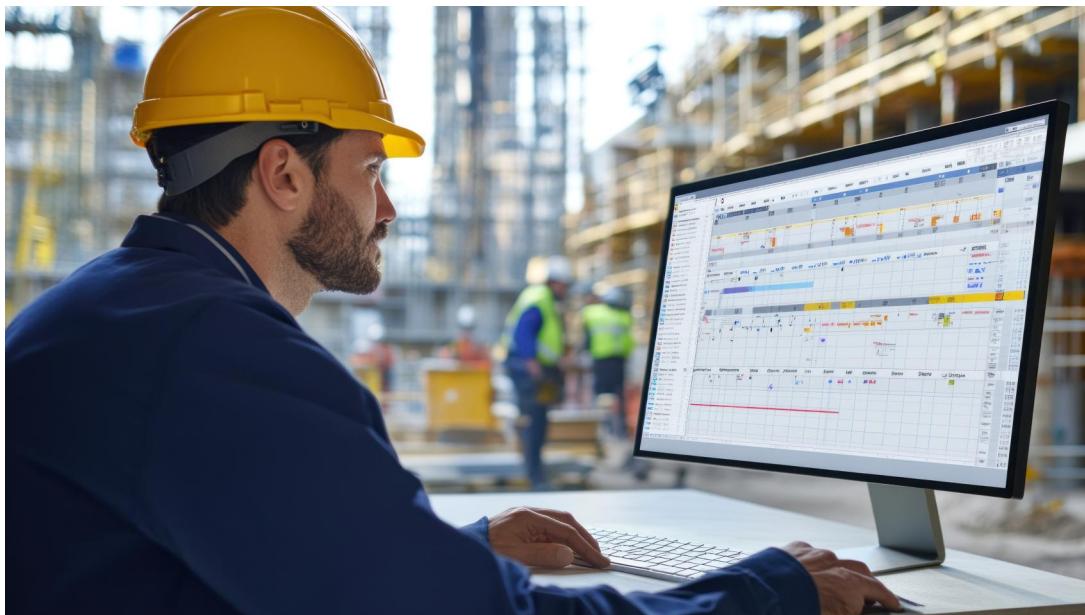


Efficient Serialization

Compression

Parallel Processing

# Optimizing Message Processing



Efficient Serialization

Compression

Parallel Processing

Prioritization

# Monitoring & Tuning Performance



Unlock the Power of Messaging Patterns

# Monitoring & Tuning Performance



Performance Metrics

# Monitoring & Tuning Performance



Performance Metrics

Logging and Tracing

# Monitoring & Tuning Performance



Performance Metrics

Logging and Tracing

Automated Alerts

# Monitoring & Tuning Performance



Performance Metrics

Logging and Tracing

Automated Alerts

Continuous Tuning

# Recap: Performance

Latency and  
Throughput

Message  
Processing

Monitoring and  
Tuning

# Security

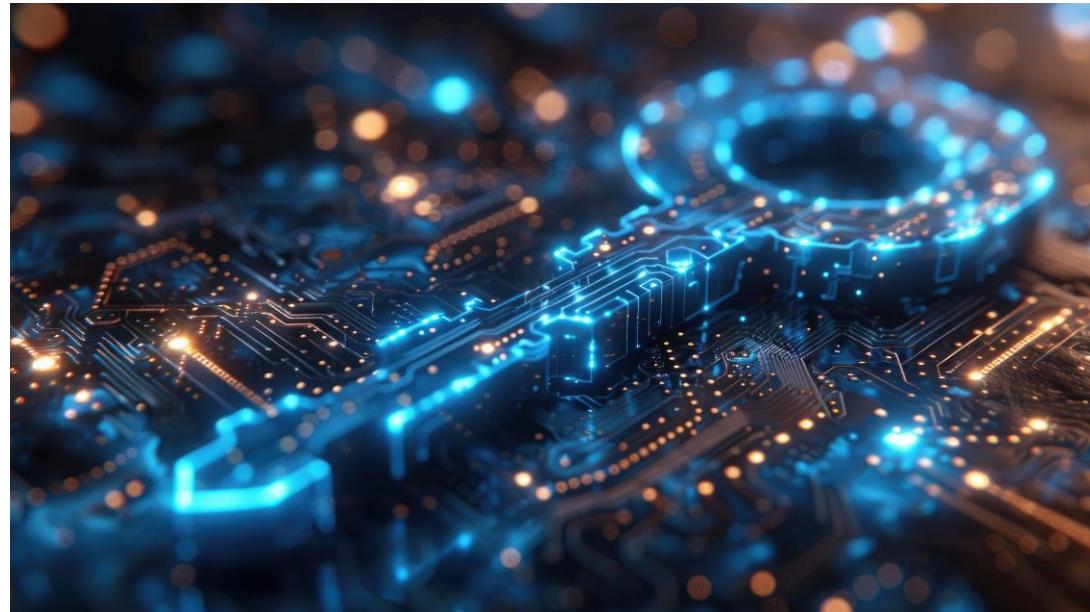
Design Considerations

Unlock the Power of Messaging Patterns

# Ensuring Secure Message Transmission and Storage

# Data Encryption

Ensuring Secure Message Transmission and Storage



TLS

Encryption at Rest

# Authentication and Authorization

Ensuring Secure Message Transmission and Storage



**OAuth and OpenID**

**Access Control Lists**

# Data Encryption and Integrity

Unlock the Power of Messaging Patterns

# Message Integrity

Data Encryption and Integrity

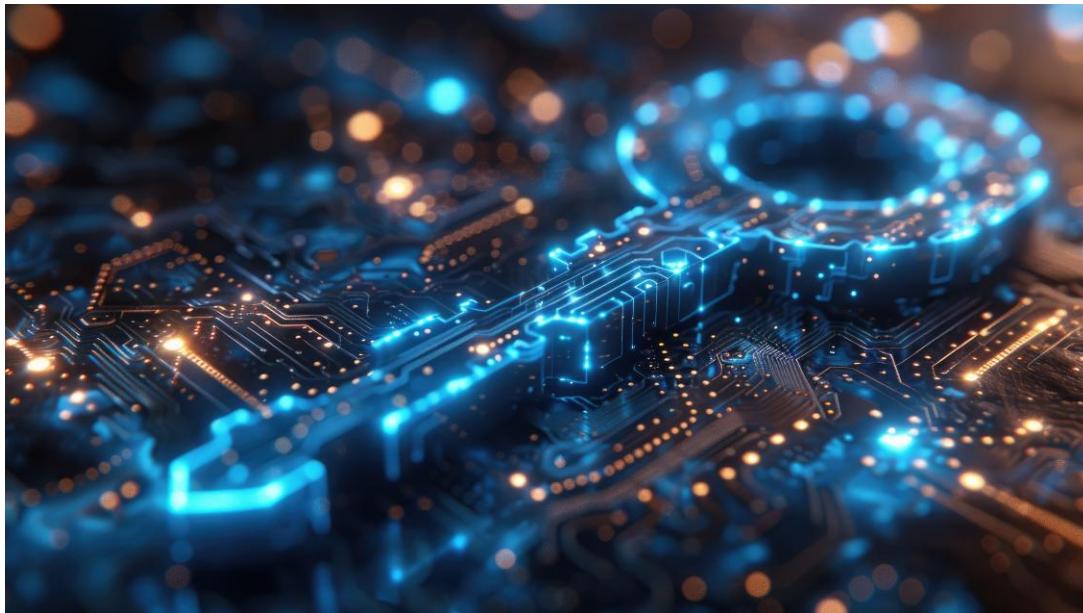


HMAC

Digital Signatures

# End-to-End Encryption

Data Encryption and Integrity



PKI

Secure Key Management

# Recap: Security

Encryption

Authentication  
and Authorization

Message Integrity

End-to-End  
Encryption

# Fault Tolerance

Design Considerations

Unlock the Power of Messaging Patterns

# Designing for Fault Tolerance and Disaster Recovery

# Biggest Outage in Years



**CROWDSTRIKE**

# Biggest Outage in Years



# Delta's Issues



Single Point of Failure

Lack of Redundancy

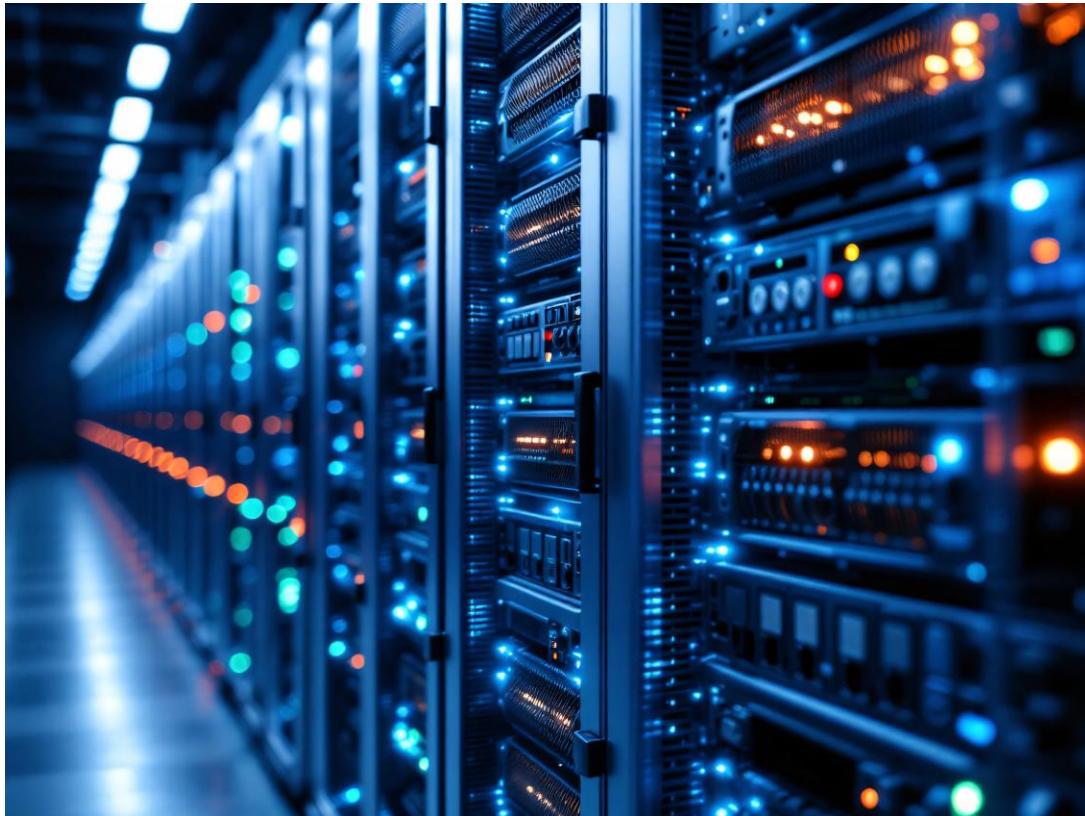
Slow Recovery Process

Communication Challenges

Operational Disruption

# Redundancy

Designing for Fault Tolerance and Disaster Recovery



Replication

Failover Mechanisms

# Disaster Recovery

Designing for Fault Tolerance and Disaster Recovery



Data Backups

Recovery Plans

# Techniques for Handling Partial Failures

# Graceful Degradation

Techniques for Handling Partial Failures



**Service Isolation**

**Fallback Strategies**

# Retry and Timeout Policies

Techniques for Handling Partial Failures



Retry Logic

Timeouts

# Strategies for Ensuring System Resilience

# Monitoring and Alerting

Techniques for Handling Partial Failures



Health Checks

Automated Alerts

# Load Balancing and Scaling

Techniques for Handling Partial Failures

Dynamic Scaling

Load Distribution

# Recap: Fault Tolerance

Redundancy and  
Disaster Recovery

Handling Partial  
Failures

System Resilience

# Flexibility and Adaptability



Design Considerations

Unlock the Power of Messaging Patterns

# Supporting Multiple Protocols and Formats

# Protocol Agnosticism

Supporting Multiple Protocols and Formats



Protocol Gateways

Pluggable Architecture

# Flexible Data Formats

Supporting Multiple Protocols and Formats



**Serialization Libraries**

**Schema Evolution**

# Handling Dynamic Workloads

Unlock the Power of Messaging Patterns

# Elastic Scaling

Handling Dynamic Workloads



**Auto-Scaling**

**Serverless Computing**

**Containerization**

# Load Shifting

Handling Dynamic Workloads



**Dynamic Load Balancing**

**Resource Reallocation**

# Adapting to Changing Requirements

# Modular Design

Adapting to Changing Requirements



Microservices Architecture

API-First Approach

# Configuration Management

Adapting to Changing Requirements

Centralized Configuration

Version Control

# Recap: Flexibility and Adaptability

Protocol and  
Format Flexibility

Dynamic  
Workloads

Changing  
Requirements

# Summary

Design Considerations

Unlock the Power of Messaging Patterns

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Summary

Scalability

Performance

Security

Fault Tolerance

Flexibility and  
Adaptability

# Design Considerations Conclusion

Building Robust Messaging Systems

# Design Considerations Conclusion

Building Robust Messaging Systems

Continuous Improvement

# Conclusion

Unlock the Power of Messaging Patterns

Unlock the Power of Messaging Patterns

# Summary of Key Takeaways

## Messaging Patterns

- Point-to-Point Messaging
- Publish/Subscribe Messaging
- Competing Consumers
- Request/Reply Messaging

# Summary of Key Takeaways

Messaging Patterns

Routing and  
Processing

- Messaging Routing
- Dead Letter Queues
- Messaging Filtering
- Aggregator Pattern
- Scatter-Gather

# Summary of Key Takeaways

Messaging Patterns

Routing and Processing

Advanced Techniques

- Idempotent Receivers
- Messaging Scheduling
- Transactional Queues
- Claim Checks

# Summary of Key Takeaways

Messaging Patterns

Routing and Processing

Advanced Techniques

Resilience and Reliability

- Circuit Breaker
- Saga
- Sequence Convoy

# Summary of Key Takeaways

Messaging Patterns

Routing and Processing

Advanced Techniques

Resilience and Reliability

Streaming Patterns

- Event Streaming

# Summary of Key Takeaways

Messaging Patterns

Routing and Processing

Advanced Techniques

Resilience and Reliability

Streaming Patterns

Design Considerations

- Reliability
- Scalability
- Performance

- Security
- Fault Tolerance
- Flexibility and Adaptability

# Questions

✉️ chadgreen@chadgreen.com

✳️ TaleLearnCode

🌐 ChadGreen.com

🐦 ChadGreen & TaleLearnCode

linkedin ChadwickEGreen