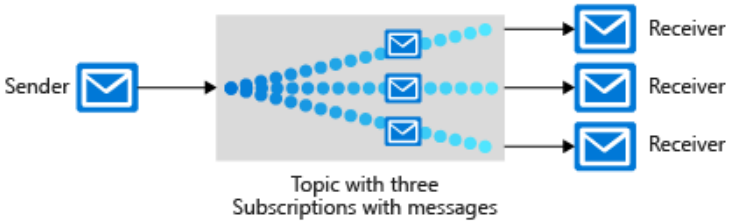
V.3 Develop message-based solutions

jeudi 16 septembre 2021 14:34

Implement solutions that use Azure Service Bus

Definition : Fully managed entreprise integration message broker. Follows AMQP protocol.



When to use :

- Your solution needs to receive messages without having to poll the queue. With Service Bus, you can achieve it by using a long-polling receive operation using the TCP-based protocols that Service Bus supports.
- Your solution requires the queue to provide a guaranteed first-in-first-out (FIFO) ordered delivery.
- Your solution needs to support automatic duplicate detection.
- You want your application to process messages as parallel long-running streams (messages are associated with a stream using the session ID property on the message). In this model, each node in the consuming application competes for streams, as opposed to messages. When a stream is given to a consuming node, the node can examine the state of the application stream state using transactions.
- Your solution requires transactional behavior and atomicity when sending or receiving multiple messages from a queue.
- Your application handles messages that can exceed 64 KB but won't likely approach the 256-KB limit.

Queues :

- FIFO message delivery. Only one consumer receives and processes each message.

Receives modes :

- Receive and delete : When SB receives the request from the consumer, it marks the message as being consumed an returns it to the consumer application.
- Peek lock :
  - Finds the next message to be consumed, locks it and then return the message to the application.
  - After the application finishes processing the message, it requests the Service Bus service to complete the second stage of the receive process. Then, the service marks the message as being consumed.

Topics and subscriptions

Unlike queues, provide one-to-many form of communication in a pub/sub pattern. Each published message is made available to each subscription registered with the topic.

Publisher sends a message to a topic and one or more subscribers receive a copy of the message, depending on filter rules set on these subscriptions. The subscriptions can use additional filters to restrict the messages that they want to receive.

Dev : Send messages to a queue

```
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;

// connection string to your Service Bus namespace
static string connectionString = "<NAMESPACE CONNECTION STRING>";

// name of your Service Bus topic
static string queueName = "az204-queue";

// the client that owns the connection and can be used to create senders and receivers
static ServiceBusClient client;

// the sender used to publish messages to the queue
static ServiceBusSender sender;

// number of messages to be sent to the queue
private const int numOfMessages = 3;

static async Task Main()
    {
        // Create the clients that we'll use for sending and processing messages.
        client = new ServiceBusClient(connectionString);
        sender = client.CreateSender(queueName);

        // create a batch
        using ServiceBusMessageBatch messageBatch = await sender.CreateMessageBatchAsync();

        for (int i = 1; i <= 3; i++)
        {
            // try adding a message to the batch
            if (!messageBatch.TryAddMessage(new ServiceBusMessage($"Message {i}")))
            {
                // if it is too large for the batch
                throw new Exception($"The message {i} is too large to fit in the batch.");
            }
        }
```

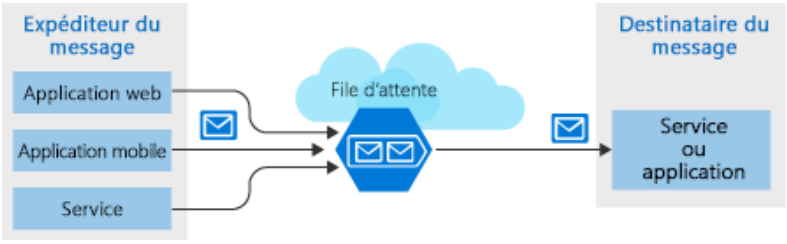
```
        try
        {
            // Use the producer client to send the batch of messages to the Service Bus queue
            await sender.SendMessagesAsync(messageBatch);
            Console.WriteLine($"A batch of {numOfMessages} messages has been published to the queue.");
        }
        finally
        {
            // Calling DisposeAsync on client types is required to ensure that network
            // resources and other unmanaged objects are properly cleaned up.
            await sender.DisposeAsync();
            await client.DisposeAsync();
        }

        Console.WriteLine("Press any key to end the application");
        Console.ReadKey();
    }
```

Dev : Receive messages from the queue

```
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;

// connection string to your Service Bus namespace
static string connectionString = "<NAMESPACE CONNECTION STRING>";

// name of your Service Bus topic
static string queueName = "az204-queue";

// the client that owns the connection and can be used to create senders and receivers
static ServiceBusClient client;

// the processor that reads and processes messages from the queue
static ServiceBusProcessor processor;

static async Task Main()
{
    // Create the client object that will be used to create sender and receiver objects
    client = new ServiceBusClient(connectionString);

    // create a processor that we can use to process the messages
    processor = client.CreateProcessor(queueName, new ServiceBusProcessorOptions());

    try
    {
        // add handler to process messages
        processor.ProcessMessageAsync += MessageHandler;

        // add handler to process any errors
        processor.ProcessErrorAsync += ErrorHandler;

        // start processing
        await processor.StartProcessingAsync();

        Console.WriteLine("Wait for a minute and then press any key to end the processing");
        Console.ReadKey();

        // stop processing
        Console.WriteLine("\nStopping the receiver...");
        await processor.StopProcessingAsync();
        Console.WriteLine("Stopped receiving messages");
    }
    finally
    {
        // Calling DisposeAsync on client types is required to ensure that network
        // resources and other unmanaged objects are properly cleaned up.
        await processor.DisposeAsync();
        await client.DisposeAsync();
    }
}

// handle received messages
static async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"Received: {body}");

    // complete the message. messages is deleted from the queue.
    await args.CompleteMessageAsync(args.Message);
}

// handle any errors when receiving messages
static Task ErrorHandler(ProcessErrorEventArgs args)
{
    Console.WriteLine(args.Exception.ToString());
    return Task.CompletedTask;
}
```

Implement solutions that use Azure Queue Storage Queue

Definition :



When to use :

- Your application must store over 80 gigabytes of messages in a queue.
- Your application wants to track progress for processing a message in the queue. It's useful if the worker processing a message crashes. Another worker can then use that information to continue from where the prior worker left off.
- You require server side logs of all of the transactions executed against your queues.

Components :

- URL format: Queues are addressable using the URL format https://<storage account>.queue.core.windows.net/<queue>. For example, the following URL addresses a queue in the diagram above https://myaccount.queue.core.windows.net/images-to-download
- Storage account: All access to Azure Storage is done through a storage account.
- Queue: A queue contains a set of messages. All messages must be in a queue. Note that the queue name must be all lowercase.
  The associated metric name is **ActiveMessageCount** (= messages that are in an active state and ready for delivery)
- Message: A message, in any format, of up to 64 KB. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

Dev

Create a queue

```
/////// Create the Queue Service client
QueueClient queueClient = new QueueClient(connectionString, queueName);

//////// Create a queue
// Get the connection string from app settings
string connectionString = ConfigurationManager.AppSettings["StorageConnectionString"];

// Instantiate a QueueClient which will be used to create and manipulate the queue
QueueClient queueClient = new QueueClient(connectionString, queueName);

// Create the queue
queueClient.CreateIfNotExists();
```

Insert a message into a queue

```
if (queueClient.Exists())
{
    // Send a message to the queue
    queueClient.SendMessage(message);
}
```

Peek at the next message : (read it without removing it from the queue)

```
if (queueClient.Exists())
{
    // Peek at the next message
    PeekedMessage[] peekedMessage = queueClient.PeekMessages();
}
```

Change the contents of a queued message

```
if (queueClient.Exists())
{
```

```
    // Get the message from the queue
    QueueMessage[] message = queueClient.ReceiveMessages();

    // Update the message contents
    queueClient.UpdateMessage(message[0].MessageId,
            message[0].PopReceipt,
            "Updated contents",
            TimeSpan.FromSeconds(60.0)  // Make it invisible for another 60 seconds
        );
}
```

Dequeue the next message

```
if (queueClient.Exists())
{
    // Get the next message
    QueueMessage[] retrievedMessage = queueClient.ReceiveMessages();

    // Process (i.e. print) the message in less than 30 seconds
    Console.WriteLine($"Dequeued message: '{retrievedMessage[0].MessageText}'");

    // Delete the message
    queueClient.DeleteMessage(retrievedMessage[0].MessageId, retrievedMessage[0].PopReceipt);
}
```

Get the queue length

```
if (queueClient.Exists())
{
    QueueProperties properties = queueClient.GetProperties();

    // Retrieve the cached approximate message count.
    int cachedMessagesCount = properties.ApproximateMessagesCount;

    // Display number of messages.
    Console.WriteLine($"Number of messages in queue: {cachedMessagesCount}");
}
```

Delete a queue

```
if (queueClient.Exists())
{
    // Delete the queue
    queueClient.Delete();
}
```