Manual del Módulo 3: El Patrón de Coordinación

Orquestando Transacciones a Prueba de Fallos con Sagas y Colas

1. Introducción a la Sesión

Bienvenido a la tercera sesión de nuestro curso. Hoy implementaremos el **Patrón Saga** en su modalidad de **Orquestación** para gestionar transacciones distribuidas, utilizando Azure Service Bus como nuestro sistema de mensajería.

2. Conceptos Teóricos Clave

- **Patrón Saga**: Una secuencia de transacciones locales para mantener la consistencia de datos entre microservicios. Si un paso falla, se ejecutan transacciones de compensación para revertir los cambios anteriores.
- **Orquestación**: Un servicio central (el Orquestador) dirige a los servicios participantes enviándoles comandos a través de colas.

3. Preparación del Entorno en Azure (Método Automatizado)

Este script de Azure CLI creará todos los recursos necesarios y configurará automáticamente la cadena de conexión en tu terminal actual.

Requisitos Previos:

- Tener una suscripción de Azure activa.
- Tener instalado el Azure CLI.
- Haber iniciado sesión en tu cuenta con `az login`.

Script de Configuración:

Abre tu terminal (bash, zsh) y ejecuta los siguientes comandos.

```
# 1. Variables de configuración
RESOURCE_GROUP="rg-curso-patrones"
LOCATION="eastus"
SERVICE_BUS_NAMESPACE="sb-curso-patrones-$RANDOM"
QUEUE_NAME="pagos-procesar-orden"
ENV_VAR_NAME="SERVICEBUS_CONNECTION_STRING"

# 2. Crear el Grupo de Recursos
echo "Creando el grupo de recursos: $RESOURCE_GROUP..."
az group create --name $RESOURCE_GROUP --location $LOCATION -o none

# 3. Crear el Namespace de Service Bus
echo "Creando el Namespace de Service Bus: $SERVICE_BUS_NAMESPACE..."
az servicebus namespace create --name $SERVICE_BUS_NAMESPACE --resource-gr

# 4. Crear la Cola dentro del Namespace
echo "Creando la cola: $QUEUE_NAME..."
az servicebus queue create --name $QUEUE_NAME --namespace-name $SERVICE_BU

# 5. Obtener la cadena de conexión y exportarla como variable de entorno
echo "Obteniendo la cadena de conexión y exportándola a una variable de en
CONN_STRING=$(az servicebus namespace authorization-rule keys list --name

# 4. Exportar la variable y mostrar el comando para otras terminales
if [ -n "$CONN_STRING" ]; then
  export "$ENV_VAR_NAME=$CONN_STRING"
  echo ""
  echo "Éxito: La variable de entorno ha sido configurada para ESTA termin
  echo ""
  echo "------------------------------------------------------------------
  echo "!! ACCIÓN REQUERIDA PARA OTRAS TERMINALES !!"
  echo "Copia la siguiente línea completa y pégala en cualquier OTRA"
  echo "terminal que necesites usar para este ejercicio:"
  echo ""
  echo "export $ENV_VAR_NAME=\"$CONN_STRING\""
  echo "------------------------------------------------------------------
  echo ""
else
  echo "Error: No se pudo obtener la cadena de conexión."
fi
```

4. Preparación del Entorno Local

1. **Clona o actualiza el repositorio:**
    - Primera vez:
      `git clone https://github.com/msouga/cursopatrones curso-patrones-dotnet`

- Actualizar: `cd curso-patrones-dotnet && git pull`

2. **NOTA IMPORTANTE:** Gracias al script de Azure, **no necesitas modificar los archivos de código C#.** Las aplicaciones leerán la configuración de la variable de entorno `SERVICEBUS_CONNECTION_STRING` que el script preparó.

5. Código Fuente Final (Referencia)

`CommandSender/Program.cs`

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

Console.WriteLine("--- Command Sender (Orquestador) ---");

var connectionString = Environment.GetEnvironmentVariable("SERVICEBUS_CONN
const string queueName = "pagos-procesar-orden";

if (string.IsNullOrEmpty(connectionString)) { /* ... mensaje de error ...

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

Console.WriteLine("Presione 'S' para simular un pedido exitoso.");
Console.WriteLine("Presione 'F' para simular un pedido fallido.");
Console.WriteLine("Presione 'Q' para salir.");

while (true)
{
    ConsoleKeyInfo key = Console.ReadKey(intercept: true);
    if (key.Key == ConsoleKey.Q) break;
    bool simulateSuccess = key.Key == ConsoleKey.S;
    var orderCommand = new ProcessOrderCommand(Guid.NewGuid(), "user-123",
    string messageBody = JsonSerializer.Serialize(orderCommand);
    var serviceBusMessage = new ServiceBusMessage(messageBody);
    await sender.SendMessageAsync(serviceBusMessage);
    Console.WriteLine($"\n[OK] Comando enviado para la orden: {orderComman
}

Console.WriteLine("\nCerrando...");
public record ProcessOrderCommand(Guid OrderId, string UserId, string Prod
```

`PaymentServiceReceiver/Program.cs`

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

```
Console.WriteLine("--- Payment Service Receiver (Worker) ---");

var connectionString = Environment.GetEnvironmentVariable("SERVICEBUS_CONN
const string queueName = "pagos-procesar-orden";

if (string.IsNullOrEmpty(connectionString)) { /* ... mensaje de error ...

await using var client = new ServiceBusClient(connectionString);
ServiceBusProcessor processor = client.CreateProcessor(queueName, new Serv
{
    AutoCompleteMessages = false,
    MaxAutoLockRenewalDuration = TimeSpan.FromMinutes(5)
});

processor.ProcessMessageAsync += MessageHandler;
processor.ProcessErrorAsync += ErrorHandler;

await processor.StartProcessingAsync();
Console.WriteLine($"Escuchando la cola '{queueName}'. Presione cualquier t
Console.ReadKey();

Console.WriteLine("\nDeteniendo el receptor...");
await processor.StopProcessingAsync();
Console.WriteLine("Receptor detenido.");

async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();
    Console.WriteLine($"\n[RECIBIDO] Mensaje. Intentando procesar...");
    try
    {
        var command = JsonSerializer.Deserialize<ProcessOrderCommand>(body
        Console.WriteLine($"--> Iniciando Saga para Orden: {command.OrderI
        Console.WriteLine("1. Validando orden... OK");
        await Task.Delay(500);
        bool stockReserved = await ReserveStock(command.ProductId);
        if (!stockReserved)
        {
            throw new InvalidOperationException($"Stock insuficiente para
        }
        await ProcessPayment(command.Amount);
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine($"--> SAGA COMPLETADA con éxito para la orden: {
        Console.ResetColor();
        await args.CompleteMessageAsync(args.Message);
    }
    catch (Exception ex)
    {
```

```
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine($"[FALLO] La saga ha fallado: {ex.Message}");
        Console.WriteLine("--> Iniciando transacciones de compensación..."
        await ReleaseStock();
        Console.WriteLine("--> Compensación finalizada.");
        Console.ResetColor();
        Console.WriteLine("--> Moviendo mensaje a la cola de mensajes fall
        await args.DeadLetterMessageAsync(args.Message, "SagaFailed", ex.M
    }
}
Task ErrorHandler(ProcessErrorEventArgs args) { /* ... */ return Task.Comp
async Task<bool> ReserveStock(string productId) { /* ... */ return product
async Task ProcessPayment(decimal amount) { /* ... */ }
async Task ReleaseStock() { /* ... */ }
public record ProcessOrderCommand(Guid OrderId, string UserId, string Prod
```

6. Ejecución y Verificación de la Demo

Instrucción Crítica: Ejecuta los siguientes comandos en la **misma sesión de terminal** (o en pestañas de la misma) donde ejecutaste el script de Azure para que la variable de entorno esté disponible.

1. **Terminal 1: Ejecuta el Receptor**
    - `cd curso-patrones-dotnet/Session3_SagaQueues/PaymentServiceReceiver`
    - `dotnet run`
2. **Terminal 2: Ejecuta el Emisor**
    - `cd curso-patrones-dotnet/Session3_SagaQueues/CommandSender`
    - `dotnet run`
3. **Pruebe los flujos:** Usa `S` para el éxito y `F` para el fallo en la terminal del emisor y observa el comportamiento correcto (sin loops) en el receptor.