# **THE OUTBOX PATTERN**

2019-03-11 [ARCHITECTURE & DESIGN](https://www.kamilgrzybek.com/blog/categories/architecture-and-design) [EVENT-DRIVEN ARCHITECTURE](https://www.kamilgrzybek.com/blog/categories/eda) [.NET](https://www.kamilgrzybek.com/blog/categories/dotnet)

Introduction

Sometimes, when processing a business operation, you need to communicate with an external component in the [Fire-and-forget](https://www.enterpriseintegrationpatterns.com/patterns/conversation/FireAndForget.html) mode. That component can be, for example:

* external service
* message bus
* mail server
* same database but different database transaction
* another database

Examples of this type of integration with external components:

* sending an e-mail message after placing an order
* sending an event about new client registration to the messaging system
* processing another *DDD Aggregate* in different database transaction - for example after placing an order to decrease number of products in stock

The question that arises is whether we are able to guarantee the [atomicity](https://en.wikipedia.org/wiki/Atomicity_(database_systems)) of our business operation from a technical point of view? Unfortunately, not always, or even if we can (using [2PC protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol)), this is a limitation of our system from the point of latency, throughput, scalability and availability. For details about these limitations, I invite you to read the article titled [It’s Time to Move on from Two Phase Commit](http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html).

The problem I am writing about is presented below:

// RegisterCustomerCommandHandler without outbox

public class RegisterCustomerCommandHandler : IRequestHandler<RegisterCustomerCommand, CustomerDto>

{

private readonly ICustomerRepository \_customerRepository;

private readonly ICustomerUniquenessChecker \_customerUniquenessChecker;

private readonly IEventBus \_eventBus;

public RegisterCustomerCommandHandler(

ICustomerRepository customerRepository,

ICustomerUniquenessChecker customerUniquenessChecker,

IEventBus eventBus)

{

this.\_customerRepository = customerRepository;

\_customerUniquenessChecker = customerUniquenessChecker;

\_eventBus = eventBus;

}

public async Task<CustomerDto> Handle(RegisterCustomerCommand request, CancellationToken cancellationToken)

{

var customer = new Customer(request.Email, request.Name, this.\_customerUniquenessChecker);

await this.\_customerRepository.AddAsync(customer);

await this.\_customerRepository.UnitOfWork.CommitAsync(cancellationToken); // commit transaction

// End of transaction--------------------------------------------------------

this.\_eventBus.Send(new CustomerRegisteredIntegrationEvent(customer.Id)); // send an event

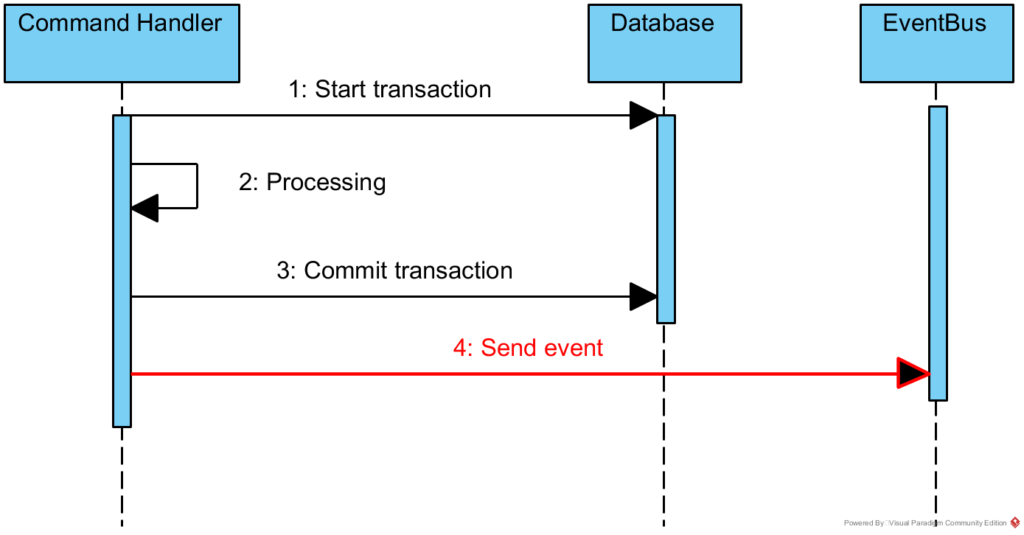
return new CustomerDto { Id = customer.Id };

}

}

After execution of **line 24** transaction is committed. In **line 28** we want to send an event to event bus, but unfortunately 2 bad things can happen:

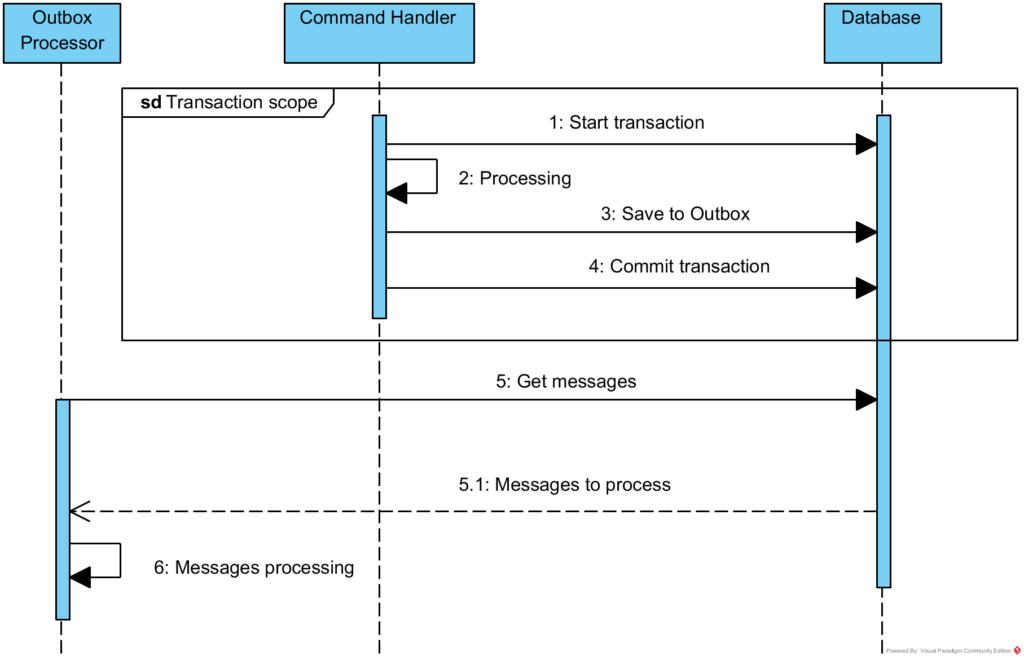
* our system can crash just after transaction commit and before sending the event
* event bus can be unavailable at this moment so the event cannot be sent



If we cannot provide atomicity or we don’t want to do that for the reasons mentioned above, what could we do to increase the reliability of our system? We should implement the *Outbox Pattern*.

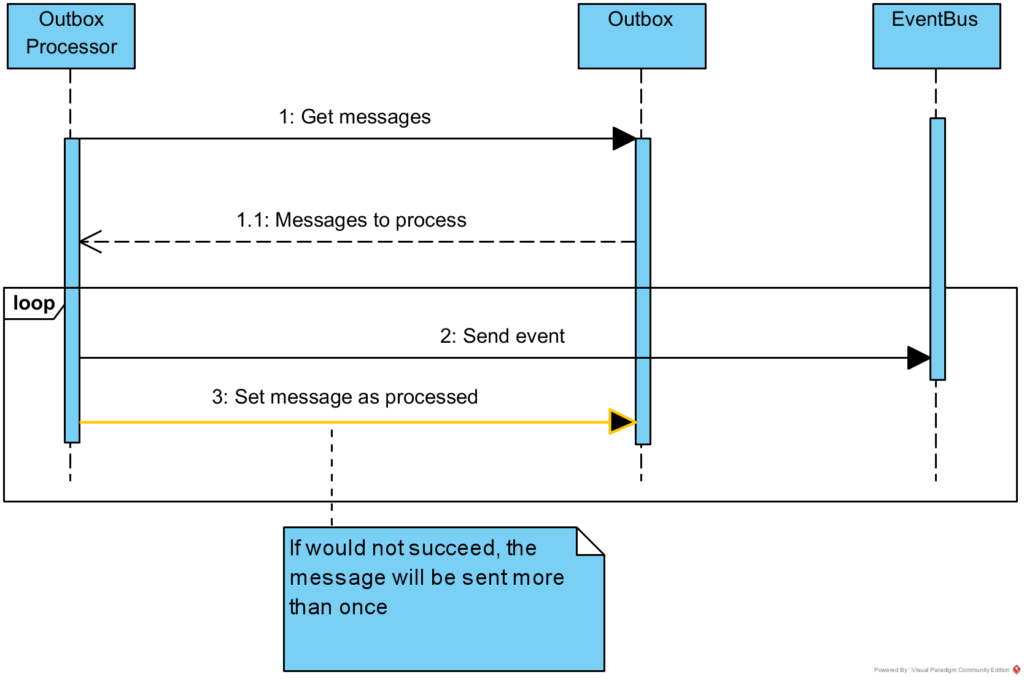
Outbox pattern

The *Outbox Pattern* is based on [Guaranteed Delivery](https://www.enterpriseintegrationpatterns.com/patterns/messaging/GuaranteedMessaging.html) pattern and looks as follows:



When you save data as part of one transaction, you also save messages that you later want to process **as part of the same transaction**. The list of messages to be processed is called an *Outbox*, just like in e-mail clients.

The second element of the puzzle is **a separate process** that periodically checks the contents of the *Outbox* and processes the messages. After processing each message, the message should be marked as processed to **avoid resending**. However, it is possible that we will not be able to mark the message as processed due to communication error with *Outbox*:



In this case when connection with Outbox is recovered, the same message will be sent again. What all this means to us? *Outbox Pattern* gives [At-Least-Once delivery](http://www.cloudcomputingpatterns.org/at_least_once_delivery/). We can be sure that message will be sent once, but can be sent multiple times too! That’s why another name for this approach is *Once-Or-More delivery*. We should remember this and try to design receivers of our messages as [Idempotents](https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html), which means:

In Messaging this concepts translates into a message that has the same effect whether it is received once or multiple times. This means that a message can safely be resent without causing any problems even if the receiver receives duplicates of the same message.

Ok, Enough theory, let’s see how we can implement this pattern in .NET world.

Implementation

Outbox message

At the beginning, we need to define the structure of our OutboxMessage:

-- Outbox messages table

CREATE SCHEMA app AUTHORIZATION dbo

GO

CREATE TABLE app.OutboxMessages

(

[Id] UNIQUEIDENTIFIER NOT NULL,

[OccurredOn] DATETIME2 NOT NULL,

[Type] VARCHAR(255) NOT NULL,

[Data] VARCHAR(MAX) NOT NULL,

[ProcessedDate] DATETIME2 NULL,

CONSTRAINT [PK\_app\_OutboxMessages\_Id] PRIMARY KEY ([Id] ASC)

)

// Outbox message

public class OutboxMessage

{

/// <summary>

/// Id of message.

/// </summary>

public Guid Id { get; private set; }

/// <summary>

/// Occurred on.

/// </summary>

public DateTime OccurredOn { get; private set; }

/// <summary>

/// Full name of message type.

/// </summary>

public string Type { get; private set; }

/// <summary>

/// Message data - serialzed to JSON.

/// </summary>

public string Data { get; private set; }

private OutboxMessage()

{

}

internal OutboxMessage(DateTime occurredOn, string type, string data)

{

this.Id = Guid.NewGuid();

this.OccurredOn = occurredOn;

this.Type = type;

this.Data = data;

}

}

// OutboxMessageEntityTypeConfiguration

internal class OutboxMessageEntityTypeConfiguration : IEntityTypeConfiguration<OutboxMessage>

{

public void Configure(EntityTypeBuilder<OutboxMessage> builder)

{

builder.ToTable("OutboxMessages", SchemaNames.Application);

builder.HasKey(b => b.Id);

builder.Property(b => b.Id).ValueGeneratedNever();

}

}

What is important, the OutboxMessage class is part of the *Infrastructure* and not the *Domain Model*! Try to talk with business about *Outbox*, they will think about the outlook application instead of the messaging pattern. :) I didn’t include ProcessedDate property because this class is only needed to save message as part of transaction so this property always will be NULL in this context.

Saving the message

For sure I do not want to program writing messages to the *Outbox* every time in each *Command Handler*, it is against [DRY principle](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself). For this reason, the *Notification Object* described in the post about [publishing Domain Events](https://www.kamilgrzybek.com/blog/posts/how-to-publish-handle-domain-events) can be used. Following solution is based on linked article with little modification - instead of processing the notifications immediately, it serializes them and writes them to the database.

As a reminder, all *Domain Events* resulting from an action are processed as part of the same transaction. If the *Domain Event* should be processed outside of the ongoing transaction, you should define a *Notification Object* for it. This is the object which should be written to the *Outbox*. The code looks like:

// Unit Of Work Commit

public async Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken))

{

var notifications = await this.\_domainEventsDispatcher.DispatchEventsAsync(this);

foreach (var domainEventNotification in notifications)

{

string type = domainEventNotification.GetType().FullName;

var data = JsonConvert.SerializeObject(domainEventNotification);

OutboxMessage outboxMessage = new OutboxMessage(

domainEventNotification.DomainEvent.OccurredOn,

type,

data);

this.OutboxMessages.Add(outboxMessage);

}

var saveResult = await base.SaveChangesAsync(cancellationToken);

return saveResult;

}

Example of *Domain Event*:

// CustomerRegisteredEvent

public class CustomerRegisteredEvent : DomainEventBase

{

public Customer Customer { get; }

public CustomerRegisteredEvent(Customer customer)

{

this.Customer = customer;

}

}

And *Notification Object*:

// CustomerRegisteredNotification

public class CustomerRegisteredNotification : DomainNotificationBase<CustomerRegisteredEvent>

{

public Guid CustomerId { get; }

public CustomerRegisteredNotification(CustomerRegisteredEvent domainEvent) : base(domainEvent)

{

this.CustomerId = domainEvent.Customer.Id;

}

[JsonConstructor]

public CustomerRegisteredNotification(Guid customerId) : base(null)

{

this.CustomerId = customerId;

}

}

First thing to note is [Json.NET](https://www.newtonsoft.com/json) library usage. Second thing to note are 2 constructors of CustomerRegisteredNotification class. First of them is for creating notification based on *Domain Event*. Second of them is to deserialize message from JSON string which is presented in following section regarding processing.

Processing the message

The processing of *Outbox* messages should take place in a separate process. However, instead of a separate process, we can also use the same process but another thread depending on the needs. Solution which is presented below can be used in both cases.

At the beginning, we need to use a scheduler that will periodically run *Outbox* processing. I do not want to create the scheduler myself (it is known and solved problem) so I will use one the mature solution in .NET - [Quartz.NET](https://www.quartz-scheduler.net/). Configuration of Quartz scheduler is very simple:

// Quartz configuration in Startup

public void StartQuartz(IServiceProvider serviceProvider)

{

this.\_schedulerFactory = new StdSchedulerFactory();

this.\_scheduler = \_schedulerFactory.GetScheduler().GetAwaiter().GetResult();

var container = new ContainerBuilder();

container.RegisterModule(new OutboxModule());

container.RegisterModule(new MediatorModule());

container.RegisterModule(new InfrastructureModule(this.\_configuration[OrdersConnectionString]));

\_scheduler.JobFactory = new JobFactory(container.Build());

\_scheduler.Start().GetAwaiter().GetResult();

var processOutboxJob = JobBuilder.Create<ProcessOutboxJob>().Build();

var trigger =

TriggerBuilder

.Create()

.StartNow()

.WithCronSchedule("0/15 \* \* ? \* \*")

.Build();

\_scheduler.ScheduleJob(processOutboxJob, trigger).GetAwaiter().GetResult();

}

Firstly, scheduler is created using factory. Then, new instance of *IoC* container for resolving dependencies is created. The last thing to do is to configure our job execution schedule. In case above it will be executed every 15 seconds but its configuration really depends on how many messages you will have in your system.

This is how ProcessOutboxJob looks like:

// ProcessOutboxJob

[DisallowConcurrentExecution]

public class ProcessOutboxJob : IJob

{

private readonly ISqlConnectionFactory \_sqlConnectionFactory;

private readonly IMediator \_mediator;

public ProcessOutboxJob(

ISqlConnectionFactory sqlConnectionFactory,

IMediator mediator)

{

\_sqlConnectionFactory = sqlConnectionFactory;

\_mediator = mediator;

}

public async Task Execute(IJobExecutionContext context)

{

using (var connection = this.\_sqlConnectionFactory.GetOpenConnection())

{

string sql = "SELECT " +

"[OutboxMessage].[Id], " +

"[OutboxMessage].[Type], " +

"[OutboxMessage].[Data] " +

"FROM [app].[OutboxMessages] AS [OutboxMessage] " +

"WHERE [OutboxMessage].[ProcessedDate] IS NULL";

var messages = await connection.QueryAsync<OutboxMessageDto>(sql);

foreach (var message in messages)

{

Type type = Assembly.GetAssembly(typeof(IDomainEventNotification<>)).GetType(message.Type);

var notification = JsonConvert.DeserializeObject(message.Data, type);

await this.\_mediator.Publish((INotification) notification);

string sqlInsert = "UPDATE [app].[OutboxMessages] " +

"SET [ProcessedDate] = @Date " +

"WHERE [Id] = @Id";

await connection.ExecuteAsync(sqlInsert, new

{

Date = DateTime.UtcNow,

message.Id

});

}

}

}

}

The most important parts are:

* Line 1 - [DisallowConcurrentExecution] attribute means that scheduler will not start new instance of job if other instance of that job is running. This is important because we don’t want process *Outbox* concurrently.
* Line 25 - Get all messages to process
* Line 30 - Deserialize message to *Notification Object*
* Line 32 - Processing the *Notification Object* (for example sending event to bus)
* Line 38 - Set message as processed

As I wrote earlier, if there is an error between processing the message (line 32) and setting it as processed (line 38), job in the next iteration will want to process it again.

Notification handler template looks like this:

// CustomerRegisteredNotificationHandler

public class CustomerRegisteredNotificationHandler : INotificationHandler<CustomerRegisteredNotification>

{

public Task Handle(CustomerRegisteredNotification notification, CancellationToken cancellationToken)

{

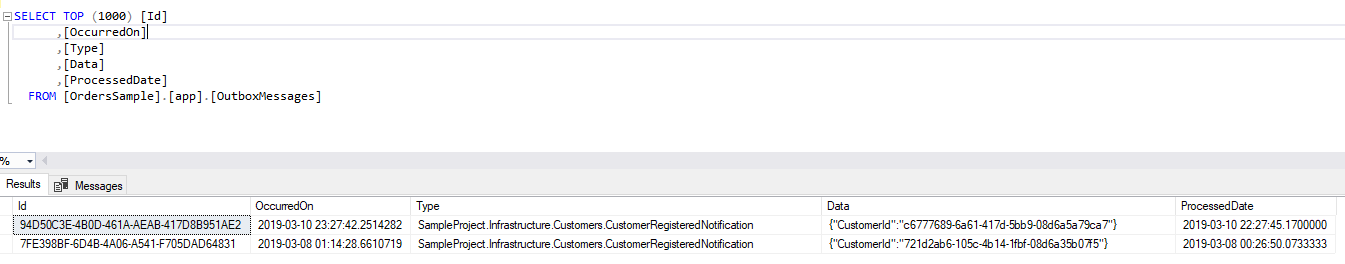
// Send event to bus or e-mail message...

return Task.CompletedTask;

}

}

Finally, this is view of our *Outbox*:



Summary

In this post I described what are the problems with ensuring the atomicity of the transaction during business operation processing. I’ve raised the topic of 2PC protocol and motivation to not use it. I presented what the *Outbox Pattern* is and how we can implement it. Thanks to this, our system can be really more reliable.

Source code

If you would like to see full, working example – check my [GitHub repository](https://github.com/kgrzybek/sample-dotnet-core-cqrs-api).

Additional Resources

1. [Refactoring Towards Resilience: Evaluating Coupling](https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/) - Jimmy Bogard
2. [Asynchronous message-based communication](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/asynchronous-message-based-communication) - Microsoft

COMMENTS