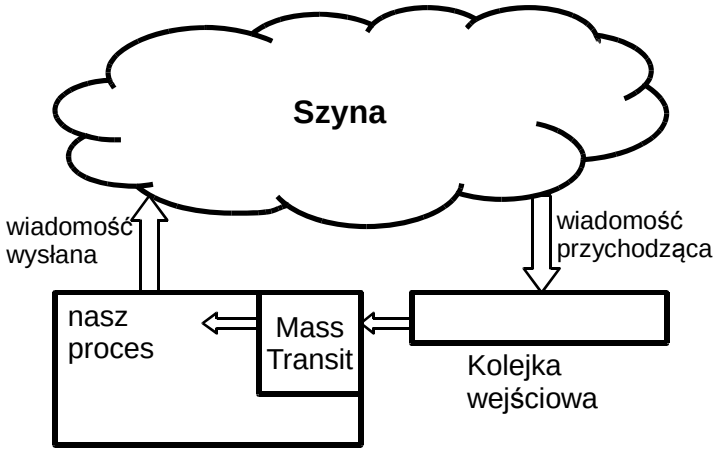
Service Bus: MassTransit
laboratorium
2019

K.M. Ocetkiewicz, T. Goluch

1. Wstęp

MassTransit[1] jest jednym z narzędzi implementujących szynę komunikacyjną. Model szyny mówi, że wszystkie urządzenia/procesy podłączone są do jednej, wspólnej szyny, dzięki której komunikują się, wymieniając komunikaty. Obiekt wysyłający „wkłada" wiadomość do szyny i nie interesuje go, kto tę wiadomość odbierze – nie specyfikuje odbiorcy, lecz wychodzi z założenia że otrzyma go ten, kogo wiadomość interesuje. Z drugiej strony odbiorca wiadomości nie zna nadawcy – przychodzi do niego interesująca wiadomość i ją obsługuje, niezależnie od tego, kto ją wysłał. MassTransit wykorzystuje MSMQ lub RabbitMQ[2] do przesyłania wiadomości.

Dokumentacja narzędzia dostępna jest pod adresem:
http://masstransit-project.com/MassTransit/



Rys. 1 Model przepływu wiadomości

Podstawowe pojęcia:

Wiadomość

Wiadomość w MassTransit jest obiektem przesyłanym przez szynę. Jej pola powinny być właściwościami, poza tym jest to zwykły obiekt.

Kolejka wejściowa

Kolejka, indywidualna dla procesu, do której trafiają komunikaty przeznaczone dla danego procesu. MassTransit monitoruje tę kolejkę i w razie pojawienia się w niej wiadomości, obsługuje ją.

Komunikacja typu wydawca/abonent (ang. publish/subscribe)

Jest to model komunikacji, w którym wydawca ogłasza wiadomości zaś abonenci je odbierają. Każdy abonent powinien zaprenumerować wiadomości wydawcy. Publikacja wiadomości powoduje przesłanie jej do wszystkich procesów, które ją zaprenumerowały. Prenumerata dotyczy konkretnych typów wiadomości – odbieramy tylko te, których typy zaprenumerowaliśmy.

2. Pobranie i instalacja

MassTransit pobieramy przez managera pakietów NuGet. Pakiet nosi nazwę MassTransit.RabbitMQ. W momencie tworzenia tej instrukcji minimalną wymaganą wersją frameworka była 4.5.2. Oczywiście serwer RabbitMQ powinien być uruchomiony, aby można było przesyłać wiadomości.

1 http://masstransit-project.com
2 https://www.rabbitmq.com/

3. Prosta komunikacja wydawca/abonent

Stwórzmy prosty komunikat:

```
namespace Komunikaty {
      public class Komunikat {
            public string tekst { get; set; }
            }
      }
```

Uwaga: aby wiadomości były poprawnie odbierane, ważne jest, by typ wiadomości u nadawcy i odbiorcy był tym samym typem. Sprowadza się to do umieszczenia komunikatu w osobnym assembly i dodaniu referencji do niego u nadawcy i odbiorcy. Jeżeli wiadomość zdefiniujemy osobno u nadawcy i odbiorcy, to nawet jeżeli nazwy, przestrzenie nazw i definicje będą identyczne, będą to różne typy i wiadomości mogą nie dochodzić do odbiorcy.

Będziemy chcieli przesłać ten komunikat pomiędzy dwoma procesami. Zacznijmy od odbiorcy:

```
...
using MassTransit;
...

public static Task Handle(ConsumeContext<Komunikaty.Komunikat> ctx) {
      return Console.Out.WriteLineAsync($"received: {ctx.Message.tekst}");
      }

static void Main(string[] args) {
      var bus = Bus.Factory.CreateUsingRabbitMq(sbc => {
            var host = sbc.Host(new Uri("rabbitmq://localhost"),
                        h => { h.Username("guest"); h.Password("guest"); });
            sbc.ReceiveEndpoint(host, "recvqueue", ep => {
                  ep.Handler<Komunikaty.Komunikat>(Handle);
                  });
            });
      bus.Start();
      Console.WriteLine("odbiorca wystartował");
      Console.ReadKey();
      bus.Stop();
      }
...
```

Rzuca się w oczy konwencja przyjęta przez twórców MassTransit – konfigurując obiekt podajemy funkcję wywołującą na otrzymanym parametrze odpowiednie metody konfiguracyjne. W naszym przypadku tworzymy szynę korzystającą z RabbitMq (Bus.Factory.CreateUsingRabbitMq). Konfigurujemy hosta RabbitMq podając jego adres („rabbitmq://localhost”) oraz dane logowanie. Szyna będzie odbierała wiadomości z kolejki o nazwie recvqueue (ReceiveEndpoint) – będzie to nasza kolejka wejściowa. Z tej kolejki obsługujemy wiadomości typu Komunikaty.Komunikat (Subscribe), obsługując odebrane wiadomości metodą Handle.

Nadawca także musi skonfigurować szynę. Dodatkowo, wyśle nasz komunikat:

```
...
using MassTransit;
...
// tworzymy i startujemy szynę jak wyżej, podając inną nazwę kolejki
Bus.Publish(new Komunikaty.Komunikat() { tekst = "asd" });
...
```

Konfiguracja jest bardzo podobna jak w przypadku odbiorcy. Nie musimy jednak podawać ReceiveEnpoint, ponieważ nic nie będziemy odbierać.

Tu warto wspomnieć zasadę dotyczącą kolejek w szynie. **Każdy proces korzystający z szyny powinien mieć swoją własną kolejkę wejściową.** Do kolejki tej będą trafiały wszystkie wiadomości przeznaczone dla naszego procesu i z niej MassTransit będzie wyciągał komunikaty, aby przekazać je do naszego kodu. Jeżeli skonfigurujemy dwa różne procesy do używania tej samej kolejki (np. równocześnie uruchomimy dwa razy odbiorcę) to w najlepszym wypadku będą one konkurować o te same komunikaty – który pierwszy odbierze, ten dostanie wiadomość. Poprawnym

rozwiązaniem jest skorzystanie z indywidualnych kolejek – wtedy każdy proces, który zaprenumeruje wiadomość, dostanie ją do własnej kolejki.

Więcej o tworzeniu szyny w różnych środowiskach można znaleźć pod adresem: http://masstransit-project.com/MassTransit/usage/configuration.html

4. Nagłówki wiadomości

Każda wiadomość może mieć dołączoną dowolną liczbę nagłówków. Nagłówek to para napisów: klucz, wartość, przy czym klucz nie powinien zawierać białych znaków, znaków interpunkcyjnych ani cyfr. Dostępne są one poprzez kontekst wiadomości. Aby je ustawić, możemy jako drugi parametr metody Publish podać funkcję otrzymującą kontekst wiadomości jako parametr i pobierając z niego pole Headers wywołać na nim metodę Set:

```
Bus.Instance.Publish(new Wiadomosc() { ... },
                     ctx => { ctx.Headers.Set("klucz", "wartosc"); }
                     );
```

Po stronie odbiorcy mamy do dyspozycji pole Headers w kontekście, będące kolekcją par (Key, Value):

```
foreach (var hdr in context.Headers.GetAll()) {
      Console.WriteLine("{0}: {1}", hdr.Key, hdr.Value);
      }
```

5. Obsługa wiadomości

Widzieliśmy już jeden ze sposobów obsługi komunikatów („Handler”): poprzez podanie funkcji z jednym parametrem (wiadomością) lub dwoma (kontekst i wiadomość). Jest to najprostszy sposób i najmniej elastyczny. Dodatkowo, sami musimy zadbać o synchronizację jeżeli może się zdarzyć, że kolejna wiadomość przyjdzie zanim wcześniejsza zostanie w pełni obsłużona.

Drugim sposobem („Instancja”) jest skorzystanie z instancji, czyli podanie konkretnego obiektu obsługującego wybrane wiadomości. Obiekt powinien implementować interfejs IConsumer<TypWiadomości>. Każdy zaimplementowany interfejs to obsługa jednego typu wiadomości, zatem pojedynczy obiekt może obsługiwać wiele ich typów. Interfejs IConsumer posiada jedną metodę:

```
public Task Consume(ConsumeContext<TypWiadomości> wiadomosc);
```

w której należy zaimplementować obsługę odebranej wiadomości. Sama wiadomość dostępna jest w polu context.Message. W przypadku takiej obsługi należy zagwarantować, że metoda Consume jest thread-safe, ponieważ może być wywołana przez wiele wątków równocześnie. Aby zarejestrować instancję obsługującą komunikaty, w trakcie ReceiveEndpoint wołamy metodę Instance podając instancję obiektu:

```
class HandlerClass: IConsumer<Komunikaty.Komunikat> { ... }
...
var instancja = new HandlerClass();
...
sbc.ReceiveEndpoint(host, "kolejka", ep => {
      ep.Instance(instancja);
      });
```

Trzeci sposób („Konsument”) obsługi wiadomości także wymaga utworzenia klasy implementującej wymieniony wcześniej interfejs. Jednak zamiast obiektu podajemy jego typ w generycznym parametrze metody Consumer, np.:

```
class HandlerClass: IConsumer<Komunikaty.Komunikat> { ... }
...
sbc.ReceiveEndpoint(host, "kolejka", ep => {
      ep.Consumer<HandlerClass>();
      });
```

Kolejną metodą jest skorzystanie z fabryki. Podajemy tu funkcję produkującą obiekty obsługujące wiadomości:

```
class HandlerZFabryki: IConsumer<msgs.Msg> {
      public Task Consume(ConsumeContext<Msg> context) { ... }
      }
...
static HandlerZFabryki Fabryka(Type typ) {
      ...
      return new HandlerZFabryki();
      }
...
endpoint.Consumer(typeof(HandlerZFabryki), Fabryka);
...
```

Parametrem fabryki jest typ obiektu, który należy utworzyć (przydatny gdy korzystamy z jednej fabryki do tworzenia obiektów różnych typów). Jeżeli mamy dedykowaną fabrykę, możemy pominąć typy:

```
...
static HandlerZFabryki Fabryka() {
      ...
      return new HandlerZFabryki();
      }
...
endpoint.Consumer(Fabryka);
...
```

Korzystając z fabryki mamy większą kontrolę nad czasem życia naszych obiektów – zamiast nowego obiektu za każdym razem, możemy zwracać zawsze ten sam, czy pobierać go z przygotowanej puli.

Konfigurując prenumeratę (endpoint) możemy podać wiele obiektów i różne metody obsługi wiadomości. Jeżeli dwoje lub więcej konsumentów/obiektów/metod zaprenumeruje tę samą wiadomość, to zostanie ona dostarczona do każdego z nich.

Prenumerata jest trwała. Nawet gdy proces abonenta nie jest uruchomiony, publikacja wiadomości spowoduje umieszczenie jej w jego kolejce wejściowej. Dzięki temu, gdy abonent wystartuje, zostaną mu dostarczone wszystkie „zaległe” wiadomości.

6. Wersjonowanie komunikatów

Jeżeli spodziewamy się, że wraz z rozwojem aplikacji przesyłane komunikaty będą ewoluować, rozsądnym rozwiązaniem jest opisanie wiadomości w postaci interfejsów. W takim przypadku w wydzielonym assembly mogą znajdować się same interfejsy (bez implementacji). Odbiorcy prenumerują te interfejsy, nadawca zaś publikuje obiekty klas, które implementują wybrane interfejsy komunikatów, np.:

```
// wiadomości
interface IMesg1 { string text { get; set; } }
interface IMesg2 { string text { get; set; } }

// odbiorca1
class Handler: IConsumer<IMesg1> {
      public Task Consume(ConsumeContext<IMesg1> msg) { ... }
      }
// odbiorca2
class Handler: IConsumer<IMesg2> {
      public Task Consume(ConsumeContext<IMesg2> msg) { ... }
      }

// nadawca
class Wiadomosc12: IMesg1, IMesg2 { ... }
Bus.Publish(new Wiadomosc12() { ... });
```

W takim przypadku zarówno odbiorca1 jak i odbiorca2 otrzymają Wiadomosc12, przy czym odbiorca1 będzie widział „fragment” IMesg1, zaś dla odbiorcy2 wiadomość będzie miała postać IMesg2.

Dzięki takiej obsłudze wiadomości, możemy wprowadzać nowe wersje komunikatów poprzez dziedziczenie. Nowa wersja rozszerza oryginalny komunikat a wydawca publikuje nową wersję. Jeżeli w systemie istnieją „stare” procesy (sprzed wprowadzenia nowej wersji komunikatu), będą one prenumerowały oryginalny interfejs, więc nadal będzie do nich docierała interesująca ich treść. „Nowi” odbiorcy (świadomi rozszerzonej wersji komunikatu) prenumerują nową wersję i „widzą” wiadomość w pełnej jej okazałości:

```
// wiadomości
interface IStaraWiadomosc { string text { get; set; } }
interface INowaWiadomosc: IStaraWiadomosc { string nowedane { get; set; } }

// stary odbiorca
class Handler: IConsumer<IStaraWiadomosc> {
      public Task Consume(ConsumeContext<IStaraWiadomosc> msg) { ... }
      }

// nowy odbiorca
class Handler: IConsumer<INowaWiadomosc> {
      public Task Consume(ConsumeContext<INowaWiadomosc> msg) { ... }
      }

// nadawca
class Wiadomosc: INowaWiadomosc { ... }
Bus.Publish(new Wiadomosc() { text=”...”, nowedane=”...” });
```

Nadawca publikuje tu tylko wiadomość nowego typu. Stary odbiorca odbiera starą „część” tej wiadomości, nowy odbiorca widzi jej całość.

7. Komunikacja dwukierunkowa

Odbiorca wiadomości ma możliwość wysłania odpowiedzi. Odpowiedź jest zwykłą wiadomością i powinna być obsłużona przez nadawcę oryginalnej wiadomości (czyli odbiorcę wiadomości zwrotnej) tak jak każda inna wiadomość przychodząca.

Odbiorca wiadomości może na nią odpowiedzieć korzystając z metody RespondAsync konteksu:

```
...
public Task Consume(ConsumeContext<Komunikaty.Komunikat> ctx) {
      ...
      ctx.RespondAsync<Komunikaty.Odpowiedz>(
                                          new Komunikaty.Odpowiedz() { ... });
      ...
      }
...
```

8. Polecenia i zdarzenia

Wiadomości w modelu publish/subscribe nazywane są w nomenklaturze MassTransita zdarzeniami. Zdarzenie zachodzi, nie mając adresata i niezależnie od tego czy ktoś na nie czeka czy nie. Tak samo publikujemy w modelu publish/subscribe – nie zastanawiamy się, kto ma być adresatem i czy w ogóle istnieje.

Polecenia z kolei są adresowane. Wydajemy polecenie konkretnemu odbiorcy i tylko jemu. Samo polecenie jest zwykłą wiadomością i obsługujemy ją tak jak w modelu publish/subscribe. Jedyna różnica pojawia się przy wysyłaniu. Po pierwsze, potrzebujemy endpoint do wysłania polecenia:

```
var tsk = bus.GetSendEndpoint(new Uri("rabbitmq://localhost/kolejka_odbiorcy"));
tsk.Wait();
var sendEp = tsk.Result;
```

następnie na endpoincie wołamy metodę Send:

```
sendEp.Send<Komunikaty.Polecenie>(new Komunikaty.Polecenie() { ... });
```

9. Wyjątki

MassTransit obsługuje wyjątki rzucane podczas obsługi wiadomości. Wiadomość, która spowodowała rzucenie wyjątku jest przesyłana do kolejki <*nazwa_kolejki_odbiorcy*>_error. Dodatkowo tworzona jest wiadomość typu Fault<T>, gdzie T jest typem wiadomości, podczas obsługi której został rzucony wyjątek. Wiadomość ta jest wysyłana na adres FaultAddress z nagłówka wiadomości (który można ustawić korzystając z kontekstu podczas wysyłania wiadomości, podobnie jak nagłówki). Jeżeli nie jest on ustawiony a jest ustawiony nagłówek ResponseAddress (można ustawić podobnie jak FaultAddress), to wiadomość wysyłana jest na ten adres. Jeżeli żaden s tych adresów nie jest podany, wiadomość jest publikowana. Wiadomości o wyjątkach obsługujemy jak zwykłe wiadomości. Komunikat typu Fault<T> ma między innymi pola Exceptions zawierające opis wyjątków oraz Message zawierające oryginalną wiadomość.

```
public static Task HndlFault(ConsumeContext<Fault<Komunikaty.Komunikat>> ctx) {
      ...
      foreach(var e in ctx.Message.Exceptions) ...
      // ctx.Message.Message = oryginalna wiadomość
      ...
      }
...
ep.Handler<Fault<Komunikaty.Komunikat>>(HndlFault);
...
```

Jeżeli wiemy, że wyjątek podczas obsługi był spowodowany chwilowym problemem, możemy zażyczyć sobie ponownego dostarczenia wiadomości. Ustawiamy to po stronie odbiorcy, korzystając z metody UseRetry podczas konfiguracji endpointu:

```
sbc.ReceiveEndpoint(host, "kolejka", ep => {
      ep.UseRetry(r => { r.Immediate(liczba); });
```

gdzie *liczba* to ilość prób doręczenia wiadomości. Powyższa konfiguracja dotyczy całego endpointu. Możemy także skonfigurować ponowne doręczanie wiadomości indywidualnie dla konsumentów:

```
ep.Consumer<Handler>(cfg =>
      { cfg.UseRetry(r =>
            { r.Interval(liczba, TimeSpan.FromMilliseconds(100)); });
      });
```

W tym przypadku żądamy *liczba* prób w odstępach co 100 milisekund (Immediate ponawia doręczenie natychmiast). Więcej o konfiguracji powtarzania doręczeń można znaleźć pod adresem (m.in. inne polityki ponawiania, filtrowanie wyjątków):
http://masstransit-project.com/MassTransit/usage/retries.html

10. Obserwowanie wiadomości

Mechanizm obserwacji w MassTransit służy do monitorowania wiadomości. Obserwacja nie jest przeznaczona do modyfikowania wiadomości. Do dyspozycji mamy kilka interfejsów:

```
interface IReceiveObserver {
      public Task PreReceive(ReceiveContext context)
            { /* wołane zaraz po odebraniu wiadomości
                 przez warstwę transportową */ }
      public Task PostReceive(ReceiveContext context)
            { /* wołane po odebraniu i obsłużeniu wiadomości */ }
      public Task PostConsume<T>(ConsumeContext<T> context,
                                 TimeSpan duration, string consumerType)
                                   where T: class
            { /* wołane po obsłużeniu wiadomości przez każdego handlera */ }
      public Task ConsumeFault<T>(ConsumeContext<T> context, TimeSpan elapsed,
                                 string consumerType, Exception exception)
                                   where T: class
            { /* wołane gdy obsługa wiadomości rzuciła wyjątek */ }
      public Task ReceiveFault(ReceiveContext context, Exception exception)
            { /* wołane gdy wyjątek wystąpił we wczesnej fazie przetwarzania
                 wiadomości (np. podczas deserializacji) */ }
      }

/* T jest typem wiadomości */
interface IConsumeObserver {
      Task IConsumeObserver.PreConsume<T>(ConsumeContext<T> context)
            { /* wołane tuż przed metodą Consume konsumenta */ }
      Task IConsumeObserver.PostConsume<T>(ConsumeContext<T> context)
            { /* wołane po metodzie Consume (o ile nie był rzucony
                 wyjątek */ }
      Task IConsumeObserver.ConsumeFault<T>(ConsumeContext<T> context,
                                            Exception exception)
            { /* wołane zamiast PostConsume w przypadku wystąpienia
                 wyjątku */ }
      }

/* T jest typem wiadomości */
interface IConsumeMessageObserver<T> where T: class {
      Task PreConsume(ConsumeContext<T> context)
            { /* wołane tuż przed metodą Consume konsumenta */ }
      Task PostConsume(ConsumeContext<T> context)
            { /* wołane po metodzie Consume (o ile nie był rzucony
                 wyjątek */ }
      Task ConsumeFault(ConsumeContext<T> context, Exception exception)
            { /* wołane zamiast PostConsume w przypadku wystąpienia
                 wyjątku */ }
      }

/* T jest typem wiadomości */
interface ISendObserver {
      public Task PreSend<T>(SendContext<T> context) where T: class
            { /* wołane tuż przed wysłaniem wiadomości, wszystkie
                 nagłówki są już ustawione */ }
      public Task PostSend<T>(SendContext<T> context) where T: class
            { /* wołane tuż po wysłaniu wiadomości */ }
      public Task SendFault<T>(SendContext<T> context, Exception exception)
            where T: class
            { /* wołane gdy wystąpił wyjątek podczas wysyłania */ }
      }
```

```
/* T jest typem wiadomości */
interface IPublishObserver {
      public Task PrePublish<T>(PublishContext<T> context) where T: class
            { /* wołane tuż przed publikacją wiadomości (tuż przed
                 wysłaniem do serwera RabbitMQ */ }
      public Task PostPublish<T>(PublishContext<T> context) where T: class
            { /* wołane tuż po wysłaniu wiadomości */ }
      public Task PublishFault<T>(PublishContext<T> context,
                                  Exception exception) where T: class
            { /* wołane gdy wystąpił wyjątek podczas publikacji */ }
      }
```

IConsumeMessageObserver jest specjalizowaną wersją IConsumeObserver. IConsumeObserver obserwuje wszystkie wiadomości, zaś IConsumeMessageObserver<T> tylko wiadomości typu T. Parametr duration informuje o czasie obsługi wiadomości, consumerType to typ klasy obsługującej wiadomość zaś exception opisuje rzucony wyjątek. Informacje dotyczące wiadomości możemy pobrać z kontekstu.

Obserwatora dołączamy do szyny, implementując odpowiedni interfejs oraz wołając na szynie odpowiednią metodę z instancją obserwatora, przed uruchomieniem szyny:

```
class ReceiveObserver: IReceiveObserver { ... }
...
var bus = Bus.Factory.CreateUsingRabbitMq( ... );
...
bus.ConnectReceiveObserver(new ReceiveObserver());
//bus.ConnectConsumeObserver (obserwator);
//bus.ConnectConsumeMessageObserver(obserwator)
//bus.ConnectSendObserver(obserwator);
//bus.ConnectPublishObserver(obserwator);
...
bus.Start();
```

Na podobnych zasadach możemy obserwować szynę. Implementując interfejs IBusObserver (http://masstransit-project.com/MassTransit/usage/lifecycle-observers.html) i podając naszą implementację jako parametr metody BusObserver szyny będziemy poinformowani o zdarzeniach takich jak uruchomienie, zatrzymanie szyny czy wystąpienie wyjątku.

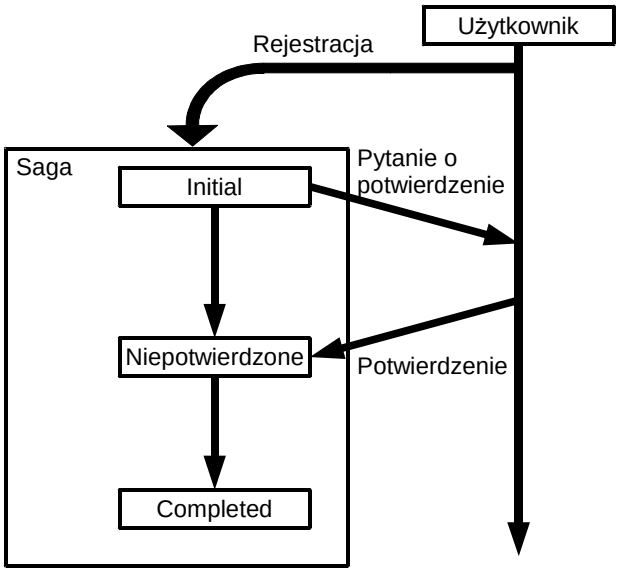
11. Sagi

Saga jest obiektem reprezentującym pewien długo trwający proces którego kolejne kroki są rozdzielone okresami nieaktywności. Przykładem może być tu rejestracja użytkownika w portalu internetowym. W pierwszym kroku użytkownik zgłasza chęć rejestracji podając login i hasło, co rozpoczyna nasz proces. System odpowiada na żądanie przygotowując pola w bazie danych i wysyłając e-mail z linkiem, który użytkownik musi kliknąć, aby potwierdzić rejestrację. Ma na to określony czas (np. 7 dni). Innym przykładem jest zakup produktu w sklepie internetowym. Najpierw użytkownik składa zamówienie. Po jego otrzymaniu sklep wysyła e-mail z prośbą o potwierdzenie zamówienia. Jeżeli użytkownik je potwierdzi, sklep wysyła treść zamówienia do centrali zarządzającej magazynem. Centrala kompletuje zamówienie i przekazuje firmie kurierskiej w celu przesłania go klientowi. Następnie firma kurierska potwierdza dostarczenie zamówienia lub zgłasza jego nieodebranie. Dopiero w tym momencie można uznać zamówienie za w pełni obsłużone (można także w ostatnim kroku wysłać klientowi ankietę do wypełnienia, czy jest zadowolony z obsługi).

Widzimy, że tego typu proces ma wiele etapów, a przejście z jednego etapu do drugiego może trwać dość długo – zanim e-mail dotrze do użytkownika, a ten kliknie na link, może minąć kilka dni, podobnie ma się sprawa z wysyłką paczki. Przez ten czas informacja o rejestracji konta lub o zamówieniu musi być pamiętana i zmieniana w reakcji na zdarzenia. Dodatkowo, w niektórych przypadkach możemy chcieć ustawić maksymalny czas życia procesu, który, jeżeli zostanie przekroczony, kończy proces niepowodzeniem, np. po 7 dniach bez reakcji użytkownika odrzucamy rejestrację, czy po kilku dniach bez potwierdzenia zamówienia anulujemy je.

Procesy takiego typu reprezentuje saga. Jest to obiekt, którego stan modyfikowany jest przez przychodzące komunikaty. W MassTransit saga jest reprezentowana przez automat skończony – obiekt, który może znajdować się w jednym z wielu dozwolonych stanów. Po utworzeniu, saga znajduje się w stanie początkowym. Następnie, pod wpływem przychodzących zdarzeń, przechodzi w inne stany aż dotrze do stanu końcowego, który ją kończy.

Implementacja sagi realizującej proces rejestracji (rysunek 2 obrazuje przepływ komunikatów) mogła by mieć postać:



Rys. 2 Przepływ komunikatów w sadze Rejestracja

Oprócz przestrzeni nazw MassTransit potrzebujemy także MassTransit.Saga oraz Automatonymous. Musimy także dodać pakiet (przez NuGet) MassTransit.Automatonymous.

```
using MassTransit;
using Automatonymous;
using MassTransit.Saga;
```

Pierwszym krokiem jest utworzenie danych sagi:

```
public class RejestracjaDane: SagaStateMachineInstance {
      public Guid CorrelationId { get; set; } // wymagane
      public string CurrentState { get; set; } // wymagane
      public string login { get; set; }
      ...
      }
```

CorrelationId przechowuje wartość identyfikującą sagę zaś CurrentState przechowuje bieżący stan sagi. Pozostałe pola tworzymy zgodnie z naszymi potrzebami. Następnie tworzymy klasę sagi:

```
public class RejestracjaSaga: MassTransitStateMachine<RejestracjaDane> {
      // stany sagi poza początkowym
      public State Niepotwierdzone { get; private set; }

      // zdarzenia
      public Event<Komunikaty.Rejestracja> Rej { get; private set; }
      public Event<Komunikaty.Potwierdzenie> Potw { get; private set; }
      ...
      }
```

Deklarujemy stany, w jakich może znaleźć się saga (pola typu State) oraz zdarzenia, na które będzie reagować saga. Zdarzenia „opakowują” komunikaty (w powyższym przykładzie zdarzenie Rej występuje w momencie odebrania komunikatu Komunikaty.Rejestracja), które wpływają na stan sagi. Komunikaty te powinny dziedziczyć po interfejsie CorrelatedBy<Guid>:

```
class Rejestracja {
      public string login { get; set; }
      ...
      }

class Potwierdzenie: CorrelatedBy<Guid> {
      public Guid CorrelationId { get; set; }
      ...
      }
```

CorrelationId komunikatu wiąże ją z sagą. Dzięki tej wartości MassTransit wie, do której sagi przekazać przychodzącą wiadomość. Jako, że Rejestracja rozpoczyna nową sagę (gdy przychodzi, nie istnieje jeszcze saga), nie posiada CorrelationId.

Musimy jeszcze opisać zachowania sagi w reakcji na komunikaty. W naszym przypadku gdy wystąpi zdarzenie Rej utworzymy nową sagę, kopiując do niej dane z wiadomości oraz wyślemy komunikat PytanieOPotwierdzenie. Gdy, będąc w stanie Niepotwierdzone, saga zaobserwuje zdarzenie Potw zakończy się. Opis zachowań sagi tworzymy w konstruktorze sagi:

```
public RejestracjaSaga() {
      InstanceState(x => x.CurrentState);
      Event(() => Rej, x => x.CorrelateBy(s => s.login,
                                          ctx => ctx.Message.login)
                              .SelectId(context => Guid.NewGuid()));
      Initially(
            When(Rej)
                  .Then(context => {
                        })
                  .ThenAsync(ctx => {
                                    return Console.Out.WriteLineAsync(
                                          $"login={ctx.Data.login}" +
                                          $"id={ctx.Instance.CorrelationId}");
                        })
                  .Respond(ctx =>
                        { return new Komunikaty.PytanieOPotwierdzenie()
                                    { id = context.Instance.CorrelationId }; })
                  .TransitionTo(Niepotwierdzone)
                  );
      During(Niepotwierdzone,
            When(Potw)
                  .Then(ctx => { Console.WriteLine("koniec"); })
                  .Finalize());
            });
      SetCompletedWhenFinalized();
      }
```

Przeanalizujmy ten kod:

```
InstanceState(x => x.CurrentState);
```
konfigurujemy tu pole przechowujące stan sagi;

```
Event(() => Rej, x => x.CorrelateBy(s => s.login,
                                    ctx => ctx.Message.login)
                              .SelectId(context => Guid.NewGuid()));
```
ponieważ zdarzenie Rej nie jest skorelowane z żadną sagą, udajemy korelację przez pole login sagi (s => s.login) z polem login wiadomości (ctx => ctx.Message.login); dodatkowo w SelectId podajemy przepis na identyfikator nowo utworzonej sagi; w ten sposób możemy także korelować wiadomości nie implementujące CorrelatedBy<Guid> z już istniejącymi sagami (pomijamy wówczas SelectId); proszę zwrócić uwagę, że pole według którego korelujemy wiadomość z sagą musi być typu referencyjnego (np. string; nie może być to typ prosty, jak int);

`During(`*`stan`*`, When(`*`zdarzenie`*`) ...`
oznacza, że definiujemy zachowanie w przypadku, gdy w stanie *stan* wystąpi zdarzenie *zdarzenie* (przyjdzie komunikat powiązany z tym zdarzeniem); do danego stanu możemy dodać klika przypadków When, podając je po przecinku: `During(Stan, When(Z1).Then(...), When(Z2).Then(...) ...`

`.Then(`*`akcja`*`), .ThenAsync(`*`akcja`*`)`
oznacza, że w danym przypadku wywołujemy akcję (asynchroniczną dla ThenAsync) podaną jako parametr do Then; parametrem jest kontekst, który w polu Data przechowuje otrzymaną wiadomość, zaś w polu Instance znajdują się dane naszej sagi;

```
.Respond(ctx => { return new Msg() { id = ctx.Instance.CorrelationId }; })
.Send(...)
.Publish(...)
```
odpowiedniki Then wysyłające komunikaty

`Initially(...)`
opisuje zachowania sagi w stanie początkowym; komunikaty w tym stanie nie są przekazywane do istniejącej sagi lecz tworzą nowe sagi; zachowuje się tak jak During, nie pobiera jednak pierwszego parametru (stanu);

`.TransitionTo(`*`stan`*`)`
oznacza, przejście do stanu *stan*,

`Finalize()`
oznacza zakończenie sagi;

`SetCompletedWhenFinalized`
oznacza, że finalizacja sagi (Finalize) powinno usunąć sagę.

Pozostaje nam jeszcze zarejestrować obsługę sagi. Robimy to podczas konfiguracji szyny, w procesie, który ma przechowywać sagi (InMemorySagaRepository<RejestracjaSaga> oznacza przechowywanie sag typu RejestracjaSaga w pamięci):

```
var repo = new InMemorySagaRepository<RejestracjaDane>();
var machine = new RejestracjaSaga();
...
var bus = Bus.Factory.CreateUsingRabbitMq(sbc => {
      ...
      sbc.ReceiveEndpoint(host, "kolejka", ep => {
            ep.StateMachineSaga(machine, repo);
            });
      });
```

Procesy wysyłające wiadomości do sagi lub odbierające od niej wiadomości konfigurujemy w tak samo, jak we wcześniejszych przypadkach.

12. Timeout sagi

Czasami saga musi zareagować na upływ czasu. Możemy to zamodelować korzystając z szeregowania komunikatów. Tworzymy komunikat informujący o upływie czasu:

```
public class Timeout: CorrelatedBy<Guid> {
      public Guid CorrelationId { get; set; }
      }
```

W danych sagi umieszczamy dodatkowe pole, identyfikujące żądanie dostarczenia opóźnionego komunikatu:

```
class Saga2Dane: SagaStateMachineInstance {
      ...
      public Guid? timeoutId { get; set; }
      }
```

W sadze opisujemy zdarzenie opisujące ten komunikat:

```
class Saga2: MassTransitStateMachine<Saga2Dane> {
      ...
      public Event<Komunikaty.Timeout> TimeoutEvt { get; private set; }
      public Schedule<Saga2Dane, Komunikaty.Timeout> TO { get; private set; }
      ...
      }
```

TO jest tu obiektem reprezentującym żądanie dostarczenia opóźnionego komunikatu. Żądanie takie konfigurujemy w konstruktorze sagi:

```
Schedule(() => TO,
          x => x.timeoutId,
          x => { x.Delay = TimeSpan.FromSeconds(10); });
```

wreszcie w odpowiednim momencie możemy wystąpić z takim żądaniem:

```
...
When(Stan)
      .Schedule(TO, ctx => new Komunikaty.Timeout()
                              { CorrelationId = ctx.Instance.CorrelationId })
```

Zdarzenie TimeoutEvt obsługujemy jak każde inne zdarzenie w sadze. Jeżeli chcemy anulować żądanie (np. użytkownik zarejestrował się w wymaganym limicie czasu), musimy zawołać:

```
...
When(Stan2)
      .Unschedule(TO)
```

Potrzebujemy jeszcze mechanizmu, który zajmie się liczeniem czasu i dostarczaniem wiadomości. Możemy tu skorzystać z pakietu MassTransit.Quartz (instalacja przez NuGet) i zawołać:

```
...
var bus = Bus.Factory.CreateUsingRabbitMq(sbc => {
      ...
      sbc.UseInMemoryScheduler();
      });
```

13. Szyfrowanie wiadomości

Jeżeli chcemy zabezpieczyć wiadomości przed nieuprawnionym odczytem, możemy je zaszyfrować. Służy do tego gotowy serializer o nazwie AesCryptoStreamProvider. Wymaga on podania dostawcy klucza implementującego interfejs ISymmetricKeyProvider. Metoda TryGetKey otrzymuje identyfikator klucza wiadomości i na tej podstawie musi wygenerować klucz szyfrujący (tablicę 32 bajtów, korzystamy z AES-256) oraz 16-bajtowy wektor inicjalizujący.

```
using MassTransit.Serialization;
...
public class Klucz: SymmetricKey {
      public byte[] IV { get; set; }
      public byte[] Key { get; set; }
      }
...
```

```
public class Dostawca: ISymmetricKeyProvider {
            private string k;
            public Dostawca(string _k) { k = _k; }
            public bool TryGetKey(string keyId, out SymmetricKey key) {
                  var sk = new Klucz();
                  sk.IV = Encoding.ASCII.GetBytes(keyId.Substring(0, 16));
                  sk.Key = Encoding.ASCII.GetBytes(k);
                  key = sk;
                  return true;
                  }
            }
...
var bus = Bus.Factory.CreateUsingRabbitMq(sbc => {
      ...
      sbc.UseEncryptedSerializer(
            new AesCryptoStreamProvider(
                  new Dostawca("0123456789abcdef0123456789abcdef"),
                              "0123456789abcdef"));
      });
...
bus.Publish(..., ctx => {
            ctx.Headers.Set(EncryptedMessageSerializer.EncryptionKeyHeader,
                        Guid.NewGuid().ToString()); });
```

Tworząc obiekt AesCryptoStreamProvider podajemy obiekt dostawcy kluczy z kluczem szyfrującym „0123456789abc def0123456789abcdef”. Klucz szyfrujący powinien być taki sam u nadawcy jak i u odbiorcy. Drugim parametrem jest domyślna wartość identyfikatora klucza wiadomości.

Wysyłając wiadomość musimy mieć możliwość przekazania odbiorcy wartości wektora inicjalizującego (nie musi być on tajny). Robimy to ostawiając nagłówek EncryptedMessageSerializer.EncryptionKeyHeader wiadomości. Wartość ta będzie widoczna zarówno u nadawcy jak i u odbiorcy w parametrze keyId metody TryGetKey. Jeżeli nie zostanie ona podana, MassTransit skorzysta z domyślnej wartości, podanej w konstruktorze AesCruptoStreamProvider.