



WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

RabbitMQ - wykorzystanie alternatywnej wymiany i wymiany martwej kolejki oraz Competing Consumers

Usługi sieciowe w biznesie

Izabela Szkaradek

Inżynieria i analiza danych, 3 rok

Numer Grupy: P4

Numer Indeksu: 166706

Rzeszów, 2023

Spis treści

1. RabbitMQ	3
1.1. Informacje	3
1.2. Podstawowe pojęcia	4
1.3. Zadanie projektu	4
2. Przygotowanie	5
2.1. Erlang	5
2.2. RabbitMQ	8
3. Projekt	10
3.1. Alternatywna wymiana i wymiana kolejki martwej	10
3.2. Competing customers	19
Bibliografia	25

1. Wstęp

1.1. Rabbit MQ - informacje

RabbitMQ jest otwartoźródłowym oprogramowaniem typu message broker, które implementuje protokół AMQP (Advanced Message Queuing Protocol). Służy do obsługi komunikacji asynchronicznej między aplikacjami.

RabbitMQ umożliwia tworzenie i zarządzanie kolejkami wiadomości, które są przekazywane między różnymi aplikacjami lub komponentami systemu. Wysyłający (producent) umieszcza wiadomość w kolejce, a odbierający (konsument) pobiera ją z kolejki w odpowiednim czasie. To pośrednictwo między producentem a konsumentem umożliwia elastyczną i skalowalną komunikację w systemach rozproszonych.

Główne funkcje RabbitMQ obejmują:

- a) Kolejowanie wiadomości - RabbitMQ umożliwia tworzenie kolejek, do których aplikacje mogą wysyłać i z których mogą odbierać wiadomości.
- b) Wieloprotokołowość - RabbitMQ obsługuje protokół AMQP, ale oferuje również interfejsy do innych protokołów komunikacyjnych, takich jak MQTT czy STOMP.
- c) Wielokrotne routowanie - Kolejki w RabbitMQ mogą być skonfigurowane do routowania wiadomości na podstawie różnych kryteriów, takich jak klucze routingu czy nagłówki wiadomości.
- d) Potwierdzanie dostarczenia - RabbitMQ zapewnia mechanizmy potwierdzania dostarczenia wiadomości, co umożliwia kontrolę nad przetwarzaniem i obsługą błędów.
- e) Elastyczność i skalowalność - Dzięki architekturze opartej na wielu węzłach, RabbitMQ może obsługiwać duże obciążenie i skalować się w zależności od potrzeb.

RabbitMQ jest stosowany w architekturach rozproszonych do obsługi komunikacji międzykomponentowej. Jego głównym zadaniem jest odbieranie, przechowywanie i dostarczanie wiadomości pomiędzy różnymi aplikacjami i serwisami.

1.2. RabbitMQ - Podstawowe pojęcia

Podstawowe pojęcia związane z RabbitMQ, które są potrzebne do głębszego zrozumienia oprogramowania:

- a) Kolejki (Queues) - kontenery, w których wiadomości są przechowywane przed przesłaniem do konsumentów, przechowują one wiadomości w kolejności FIFO (First-In-First-Out), czyli pierwsza wiadomość umieszczona w kolejce jest pierwsza do pobrania przez konsumenta. Natomiast każda kolejka ma unikalną nazwę, która jest wykorzystywana do adresowania jej w systemie RabbitMQ.
- b) Producent (Producer) - aplikacja lub komponent, który generuje wiadomości i wysyła je do RabbitMQ w celu umieszczenia ich w odpowiedniej kolejce. Jest on również odpowiedzialny za określenie kolejki, do której ma zostać wysłana wiadomość, oraz za przekazanie samej treści wiadomości.
- c) Konsument (Consumer) - aplikacja lub komponent, który pobiera wiadomości z kolejek, rejestruje się on do konkretnej kolejki, aby odbierać wiadomości. Gdy wiadomość zostaje pobrana przez konsumenta, zostaje ona usunięta z kolejki.
- d) Routing - proces określania, do której kolejki powinna zostać przekazana wiadomość na podstawie reguł zdefiniowanych przez wymiany, może być oparty m.in. na kluczach routingu, nagłówkach wiadomości, wzorcach routingu.
- e) Potwierdzanie dostarczenia (Acknowledgements) - mechanizm, który pozwala na potwierdzenie, że wiadomość została przetworzona i dostarczona do konsumenta. Po odbiorze i przetworzeniu wiadomości, konsument wysyła potwierdzenie do RabbitMQ, informując o sukcesie przetwarzania. Potwierdzenie dostarczenia zapewnia niezawodność dostarczania wiadomości.

1.3. Opis problemu

Zadaniem projektu jest zrozumienie działania oprogramowania RabbitMQ, jak i zastosowanie przykładu użycia w praktyce. Celem jest zrozumienie działania alternatywnej wymiany oraz wymiany martwej kolejki (dead-letter exchange), jak i również sposób wykonywania przykładu Competing Consumers.

2. Przygotowanie

Przed rozpoczęciem praktycznej części projektu, należało pobrać i zainstalować Erlang oraz RabbitMQ.

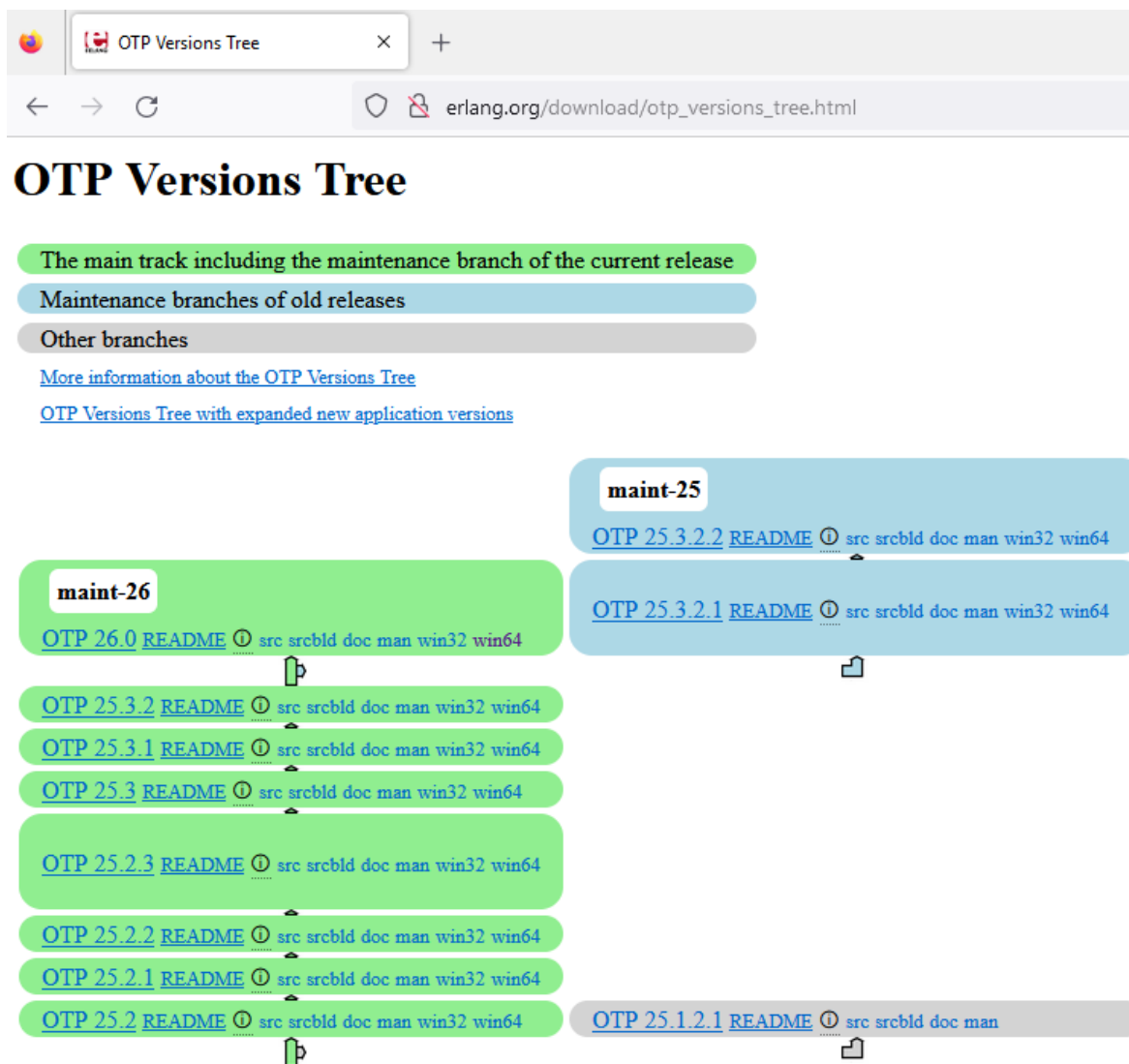
2.1. Erlang

Erlang, inaczej OTP jest otwartą platformą do budowy skalowalnych i niezawodnych systemów. Erlang w RabbitMQ pełni rolę języka programowania oraz platformy wykonawczej. RabbitMQ to rozbudowany system oprogramowania dla brokera wiadomości (message broker), który wykorzystuje język Erlang do implementacji swojej infrastruktury.

Wykorzystanie Erlanga w RabbitMQ zapewnia kilka korzyści. Spośród nich można znaleźć takie jak:

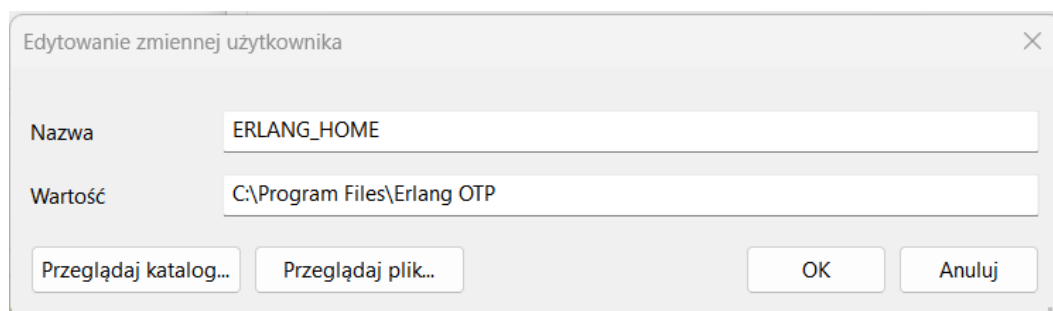
- a) Elastyczność: Erlang umożliwia łatwe rozwijanie i modyfikowanie systemu RabbitMQ. Można dodawać nowe funkcjonalności, dostosowywać zachowanie brokera wiadomości i rozszerzać jego możliwości.
- b) Niezawodność: Erlang jest projektowany do budowy systemów o wysokiej niezawodności. Posiada wbudowane mechanizmy obsługi błędów, odtwarzania systemu po awarii oraz zarządzania wątkami. Dzięki temu RabbitMQ może zapewnić wysoką dostępność i niezawodność swojego brokera wiadomości.
- c) Wydajność i skalowalność: Erlang jest znany z wysokiej wydajności i możliwości obsługi dużej liczby równoległych połączeń. Dzięki temu RabbitMQ może obsługiwać dużą liczbę wiadomości jednocześnie.
- d) Zarządzanie rozproszonymi systemami: Erlang zapewnia narzędzia i biblioteki do zarządzania komunikacją międzyprocesową, wątkami, skalowalnością i równoważeniem obciążenia. To sprawia, że RabbitMQ jest idealnym narzędziem do budowy rozproszonych systemów opartych na przesyłaniu wiadomości.

W celu uruchomienia RabbitMQ należało pobrać ERLANG, który umożliwia tworzenie rozproszonych, równoległych i odpornych na błędy systemów.



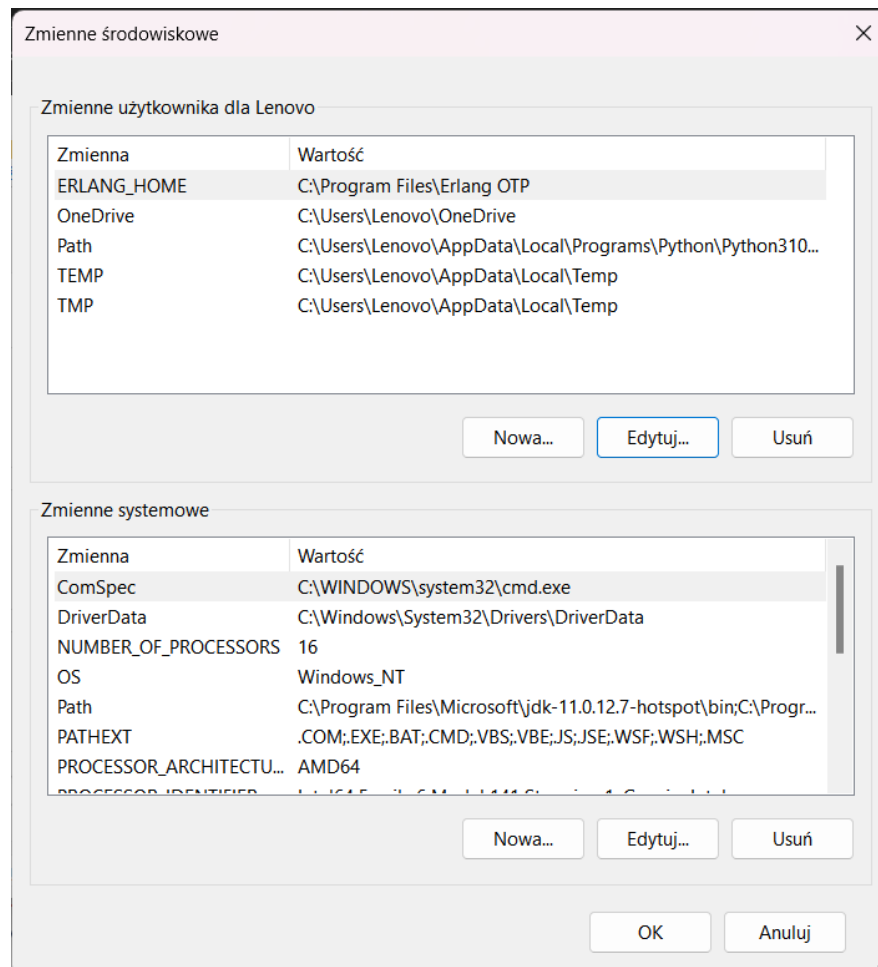
Rysunek 2.2: Strona internetowa, z której można pobrać erlang.

Po procesie instalacji, należało dodać Zmienną środowiskową (którą można znaleźć w zaawansowanych właściwościach systemu).



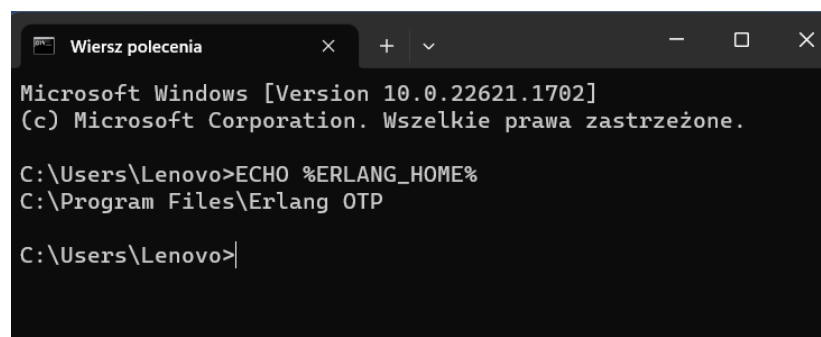
Rysunek 2.3: Dodana zmienna środowiskowa.

Po jej dodaniu, zmienne środowiskowe wyglądały następująco:



Rysunek 2.4: Zmienne środowiskowe.

Następnie, po sprawdzeniu w CMD dodanej zmiennej środowiskowej otrzymujemy jej ścieżkę, oznacza to, że wszystko przebiegło pomyślnie.



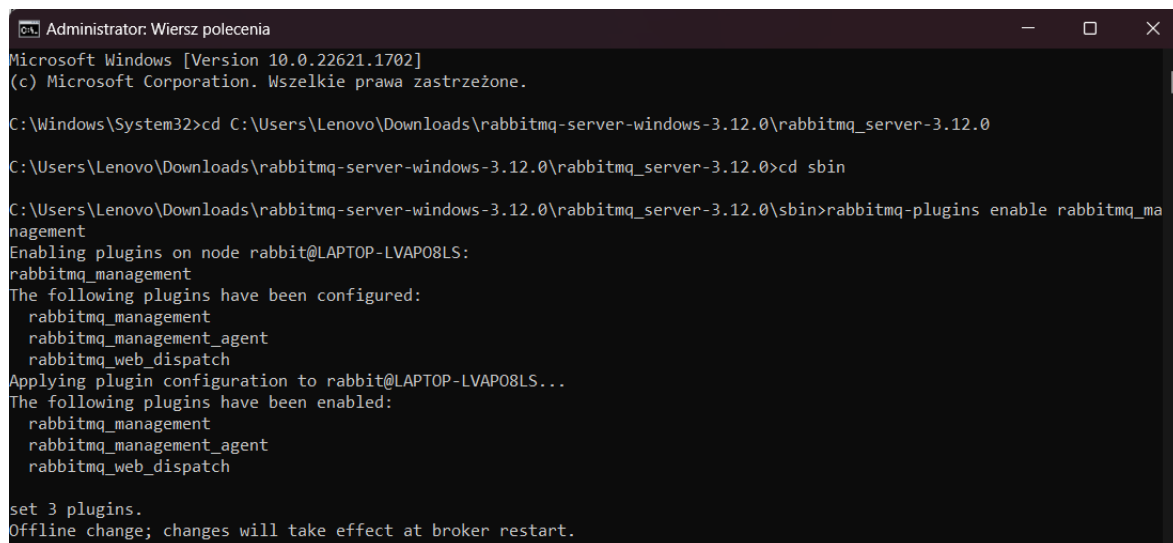
Rysunek 2.5: Sprawdzenie w CMD.

2.2. RabbitMQ

Należało pobrać RabbitMQ ze strony github, a następnie zainstalować.

Po udanej instalacji, trzeba było utworzyć wiersz poleceń z uprawnieniami administratora, a następnie wejść w ścieżkę nowo pobranego programu. Jako że przy poprzednich próbach był problem z uruchomieniem *rabbitmq-server.bat*, należało użyć polecenia *rabbitmq plugins enable rabbitmq management*, które pozwoliło na dostęp do wtyczek.

Wtyczki, które zostały włączone to: *rabbitmq management*, *rabbitmq management agent* oraz *rabbitmq web dispatch*.



```
Administrator: Wiersz polecenia
Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Windows\System32>cd C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0

C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0>cd sbin

C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0\sbin>rabbitmq-plugins enable rabbitmq_ma
nagement
Enabling plugins on node rabbit@LAPTOP-LVAP08LS:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@LAPTOP-LVAP08LS...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.
```

Rysunek 2.6: Pozwolenie na dostęp do wtyczek.

Następnie uruchomiono RabbitMQ używając polecenia *rabbitmq-server.bat*:

```
Administrator: Wiersz polecenia - rabbitmq-server.bat
Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Windows\System32>cd C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0

C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0>cd sbin

C:\Users\Lenovo\Downloads\rabbitmq-server-windows-3.12.0\rabbitmq_server-3.12.0\sbin>rabbitmq-server.bat
2023-06-07 20:54:32.157000+02:00 [notice] <0.44.0> Application syslog exited with reason: stopped
2023-06-07 20:54:32.199000+02:00 [notice] <0.235.0> Logging: switching to configured handler(s); following messages may
not be visible in this log output

## ##      RabbitMQ 3.12.0
## ##
##### Copyright (c) 2007-2023 VMware, Inc. or its affiliates.
##### ##
##### Licensed under the MPL 2.0. Website: https://rabbitmq.com

Erlang:      26.0 [jit]
TLS Library: OpenSSL - OpenSSL 3.1.0 14 Mar 2023
Release series support status: supported

Doc guides:  https://rabbitmq.com/documentation.html
Support:     https://rabbitmq.com/contact.html
Tutorials:   https://rabbitmq.com/getstarted.html
Monitoring:  https://rabbitmq.com/monitoring.html

Logs: <stdout>
      c:/Users/Lenovo/AppData/Roaming/RabbitMQ/log/rabbit@LAPTOP-LVAP08LS.log

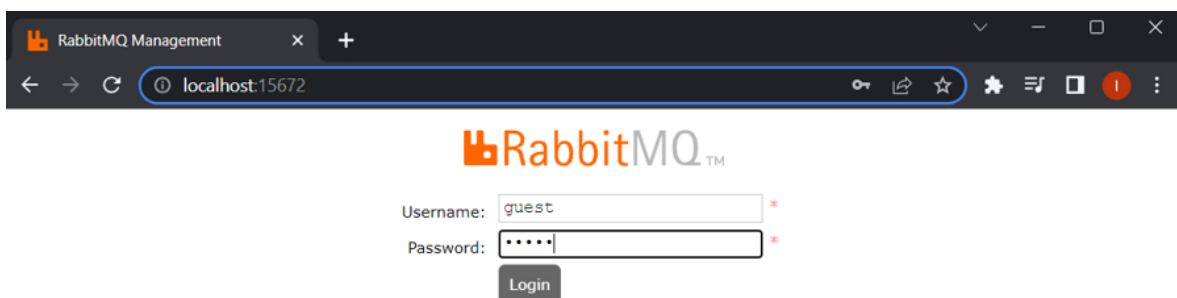
Config file(s): (none)

Starting broker... completed with 3 plugins.
```

Rysunek 2.7: Uruchomienie brokera.

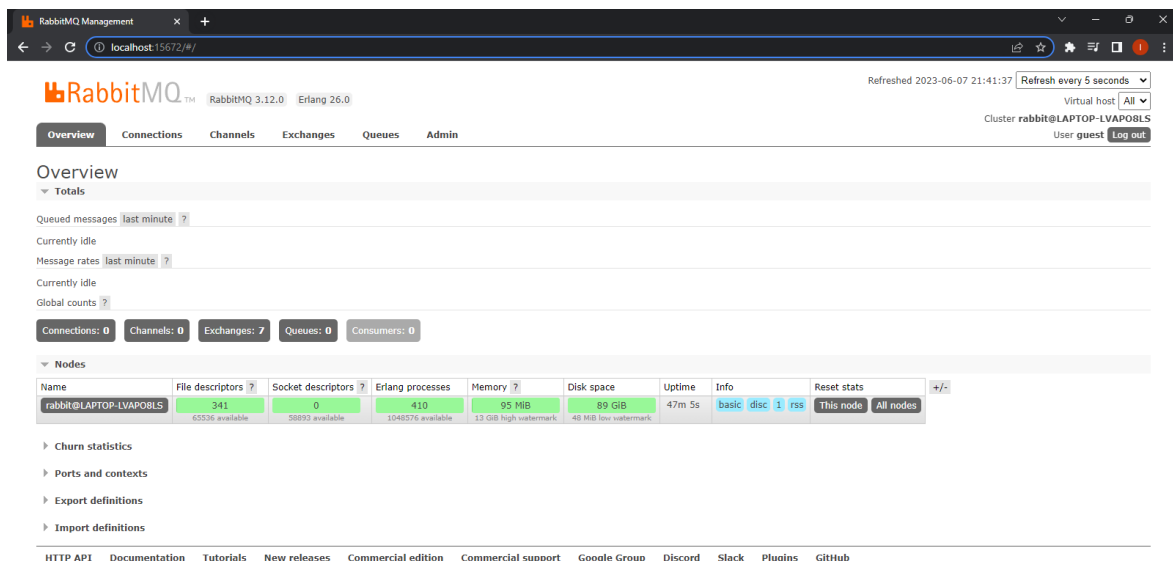
Wszystko przebiegło pomyślnie biorąc pod uwagę fakt, że jest podana informacja o 3 załadowanych wtyczkach.

Dodatkowo, po uruchomieniu *localhost:15672* w przeglądarce, otrzymano:



Rysunek 2.8: Logowanie do RabbitMQ.

Po zalogowaniu się otrzymujemy:

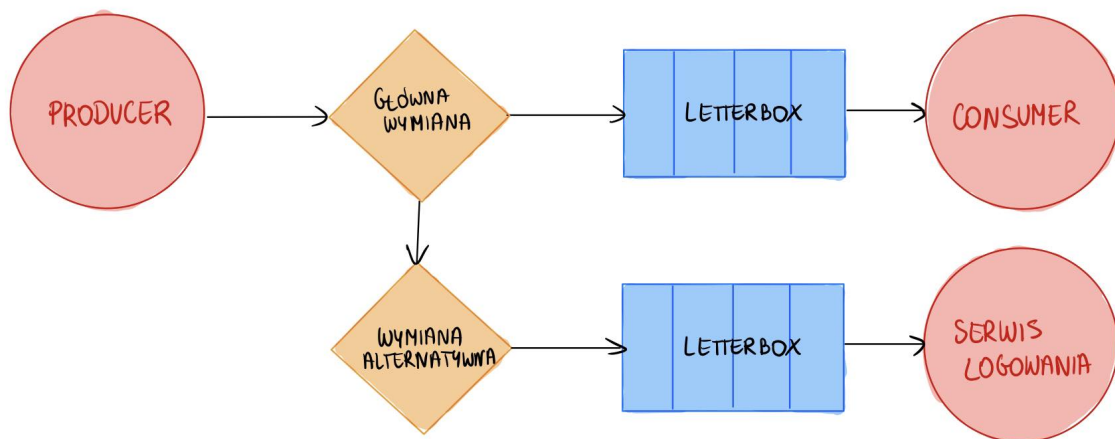


Rysunek 2.9: Strona RabbitMQ.

3. Projekt

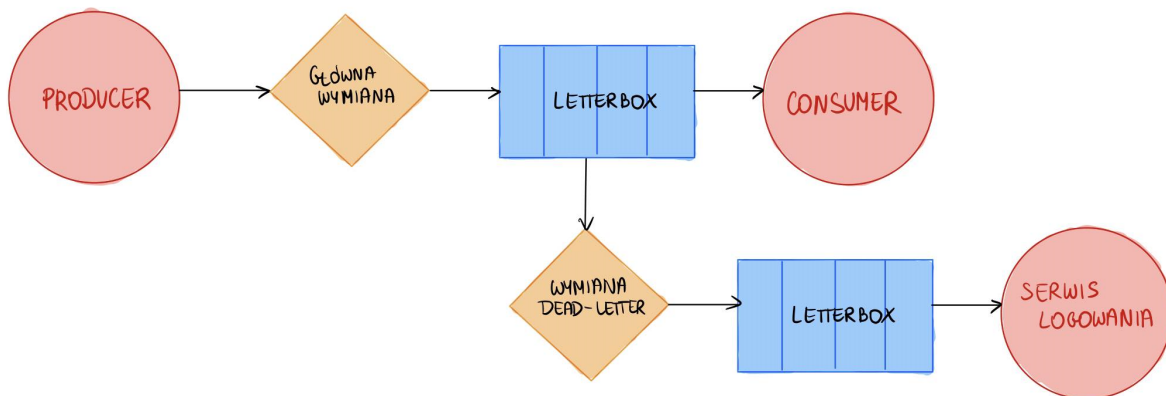
3.1. Alternatywna wymiana i wymiana kolejki martwej

Alternatywna wymiana to mechanizm umożliwiający przekierowanie wiadomości, które nie mogą zostać dostarczone do ich docelowej kolejki. Gdy wiadomość nie może zostać dostarczona do żadnej kolejki związaną bezpośrednio z daną wymianą, RabbitMQ przekierowuje tę wiadomość do alternatywnej wymiany (alternate exchange).



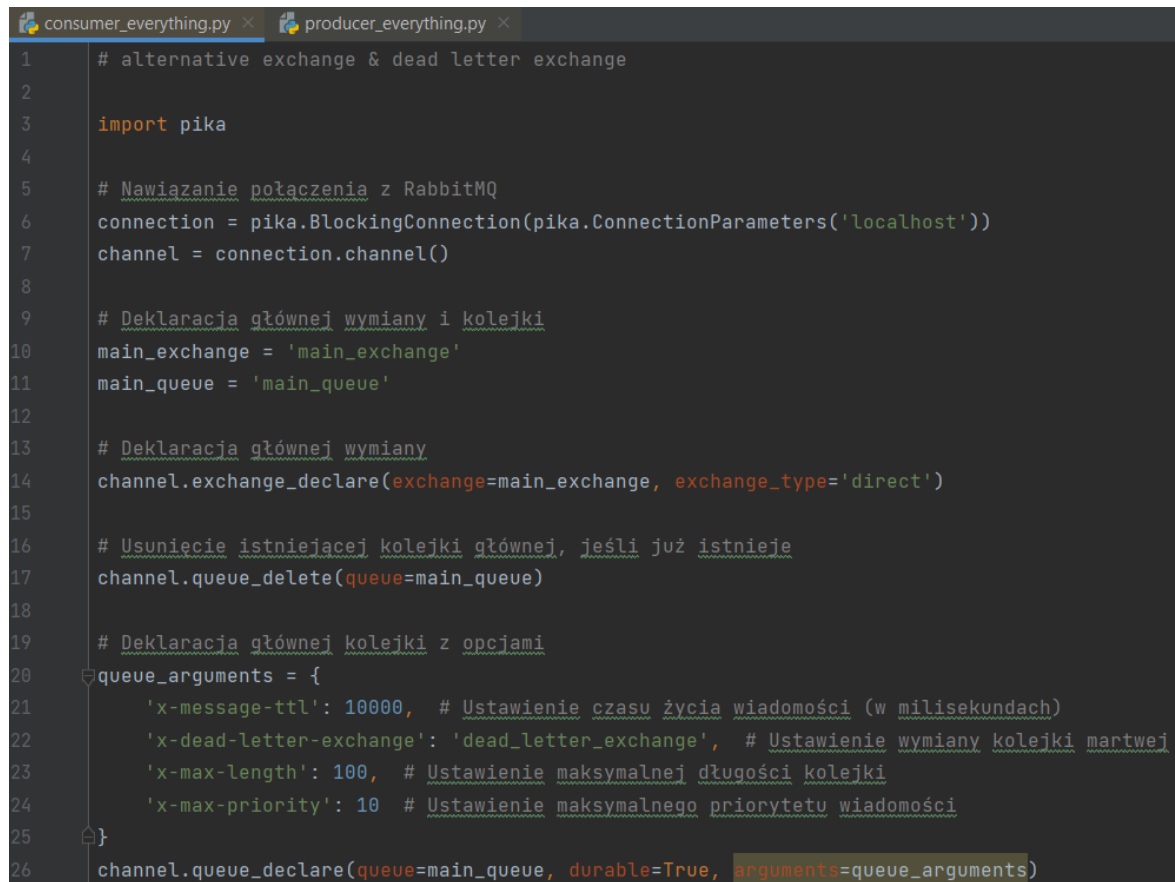
Rysunek 3.10: Alternatywna wymiana.

Wymiana kolejki martwej jest specjalnym rodzajem wymiany, która służy do przekierowywania wiadomości, które nie mogą być przetworzone przez ich pierwotną kolejkę. Gdy wiadomość zostaje oznaczona jako "umieszczona w kolejkę martwą" (ang. dead-lettered), jest przekierowywana do wymiany "dead-letter", gdzie może być dalej przetwarzana lub analizowana.



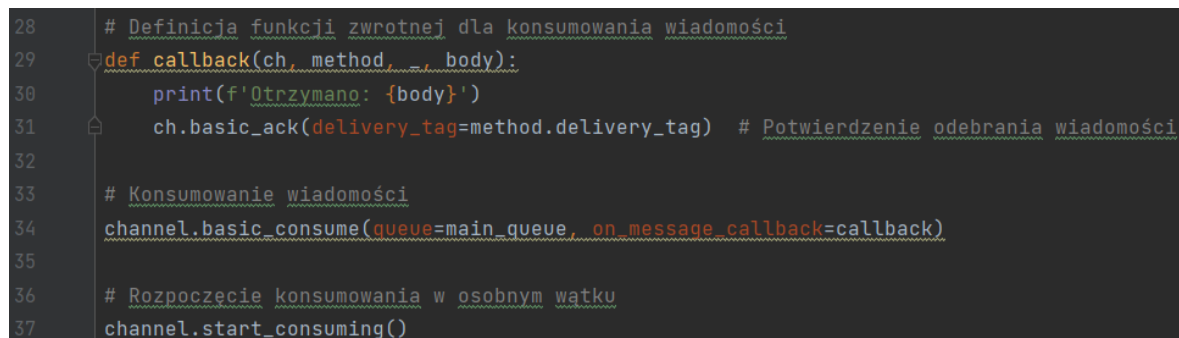
Rysunek 3.11: Wymiana kolejki martwej.

Postanowiłam połączyć te dwie wymiany, w wyniku czego stworzyłam następujący kod dla konsumenta (consumer):



```
1 # alternative exchange & dead letter exchange
2
3 import pika
4
5 # Nawiązanie połączenia z RabbitMQ
6 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
7 channel = connection.channel()
8
9 # Deklaracja głównej wymiany i kolejki
10 main_exchange = 'main_exchange'
11 main_queue = 'main_queue'
12
13 # Deklaracja głównej wymiany
14 channel.exchange_declare(exchange=main_exchange, exchange_type='direct')
15
16 # Usunięcie istniejącej kolejki głównej, jeśli już istnieje
17 channel.queue_delete(queue=main_queue)
18
19 # Deklaracja głównej kolejki z opcjami
20 queue_arguments = {
21     'x-message-ttl': 10000, # Ustawienie czasu życia wiadomości (w milisekundach)
22     'x-dead-letter-exchange': 'dead_letter_exchange', # Ustawienie wymiany kolejki martwej
23     'x-max-length': 100, # Ustawienie maksymalnej długości kolejki
24     'x-max-priority': 10 # Ustawienie maksymalnego priorytetu wiadomości
25 }
26 channel.queue_declare(queue=main_queue, durable=True, arguments=queue_arguments)
```

Rysunek 3.12: Consumer - wymiana alternatywna i kolejki martwej, część 1.



```
28 # Definicja funkcji zwrotnej dla konsumowania wiadomości
29 def callback(ch, method, _, body):
30     print(f'Otrzymano: {body}')
31     ch.basic_ack(delivery_tag=method.delivery_tag) # Potwierdzenie odebrania wiadomości
32
33 # Konsumowanie wiadomości
34 channel.basic_consume(queue=main_queue, on_message_callback=callback)
35
36 # Rozpoczęcie konsumowania w osobnym wątku
37 channel.start_consuming()
```

Rysunek 3.13: Consumer - wymiana alternatywna i kolejki martwej, część 2.

Celem tego kodu jest utworzenie prostego programu konsumenta w RabbitMQ. Program ten ma za zadanie połączyć się z lokalnym serwerem RabbitMQ, zadeklarować wymianę i kolejkę, skonfigurować odpowiednie opcje dla kolejki, skonsumentować wiadomości z kolejki oraz przetworzyć je przy użyciu określonej funkcji zwrotnej.

Następujący kod konsumenta po kolei:

- 1) Nawiązuje połączenie z lokalnym serwerem RabbitMQ.
- 2) Deklaruje wymianę o nazwie *main exchange* typu *direct*.
- 3) Usuwa istniejącą kolejkę o nazwie *main queue*, jeśli taka istnieje.
- 4) Deklaruje kolejkę o nazwie *main queue* z określonymi opcjami, takimi jak czas życia wiadomości, wymiana kolejki martwej, maksymalna długość kolejki i maksymalny priorytet wiadomości.
- 5) Definiuje funkcję zwrotną callback, która jest wywoływana przy odbiorze wiadomości z kolejki.
- 6) Konsumuje wiadomości z kolejki *main queue*, wywołując funkcję zwrotną callback dla każdej otrzymanej wiadomości.
- 7) Rozpoczyna proces konsumowania wiadomości w osobnym wątku.
- 8) Zamyka połączenie z RabbitMQ po zakończeniu konsumowania.

Natomiast niżej jest przedstawiony kod dla producenta:

```
consumer_everything.py x producer_everything.py x
1 # alternative exchange & dead letter exchange
2 import pika
3
4 # Nawiązanie połączenia z RabbitMQ
5 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
6 channel = connection.channel()
7
8 # Deklaracja głównej wymiany i kolejki
9 main_exchange = 'main_exchange'
10 main_queue = 'main_queue'
11
12 # Deklaracja głównej wymiany
13 channel.exchange_declare(exchange=main_exchange, exchange_type='direct')
14
15 # Deklaracja głównej kolejki z opcjami
16 queue_arguments = {
17     'x-message-ttl': 10000, # Ustawienie czasu życia wiadomości (w milisekundach)
18     'x-dead-letter-exchange': 'dead_letter_exchange', # Ustawienie wymiany kolejki martwej
19     'x-max-length': 100, # Ustawienie maksymalnej długości kolejki
20     'x-max-priority': 10 # Ustawienie maksymalnego priorytetu wiadomości
21 }
22 channel.queue_declare(queue=main_queue, durable=True, arguments=queue_arguments)
```

Rysunek 3.14: Producent - wymiana alternatywna i kolejki martwej, część 1.

```
24 # Połączenie kolejki głównej z główną wymianą
25 channel.queue_bind(exchange=main_exchange, queue=main_queue)
26
27 # Publikowanie wiadomości
28 message_count = 10
29 for _ in range(message_count):
30     for i in range(1, message_count+1):
31         message = f'Wiadomość {i}'
32         channel.basic_publish(
33             exchange=main_exchange,
34             routing_key='',
35             body=message.encode(), # Zakodowanie wiadomości jako bajty
36             properties=pika.BasicProperties(delivery_mode=2) # Ustawienie trwałości wiadomości
37         )
38         print(f'Opublikowano: {message}')
39
40 # Zwiększenie liczby wiadomości dla kolejnej iteracji
41 message_count *= 10
42
43 # Zamknięcie połączenia
44 connection.close()
```

Rysunek 3.15: Producent - wymiana alternatywna i kolejki martwej, część 2.

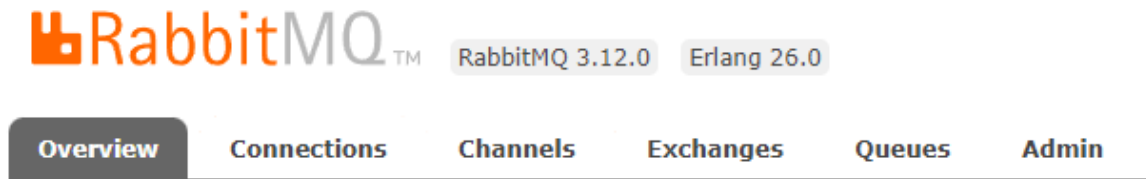
Celem tego kodu jest zaprezentowanie przykładu użycia mechanizmów alterna-

tywnej wymiany oraz kolejki martwej w RabbitMQ. Kod tworzy główną wymianę, kolejkę i publikuje wiadomości, które są przetwarzane zgodnie z zdefiniowanymi opcjami, takimi jak czas życia wiadomości, priorytet i maksymalna długość kolejki. Wykorzystanie alternatywnej wymiany pozwala na przekierowanie wiadomości, które nie mogą zostać dostarczone do głównej kolejki, do innej wymiany. Natomiast ustawienie kolejki martwej umożliwia przechwycenie wiadomości, które nie mogą zostać obsłużone przez główną kolejkę, umożliwiając dalsze przetwarzanie lub analizę takich wiadomości.

Następujący kod konsumenta po kolei:

- 1) Nawiązuje połączenie z serwerem RabbitMQ.
- 2) Deklaruje wymianę o nazwie *main exchange* typu *direct*.
- 3) Deklaruje kolejkę o nazwie *main queue* z określonymi opcjami, takimi jak czas życia wiadomości, wymiana kolejki martwej, maksymalna długość kolejki i maksymalny priorytet wiadomości.
- 4) Łączy kolejkę *main queue* z wymianą *main exchange*.
- 5) Publikuje wiadomości do wymiany *main exchange* o różnych treściach i priorytetach.
- 6) Liczba wiadomości, które są publikowane, jest iteracyjnie zwiększana, aby zobaczyć wpływ na kolejkę.
- 7) Każda wiadomość jest kodowana jako bajty i ustawiana jako trwała, aby przetrwała restart serwera RabbitMQ.
- 8) Wyświetla informacje o opublikowanych wiadomościach.
- 9) Po zakończeniu publikowania, zamyka połączenie z RabbitMQ.

Po uruchomieniu powyższego kodu możemy skorzystać z poniższych przycisków w RabbitMQ.



Rysunek 3.16: Menu RabbitMQ.

Po wybraniu "connections" otrzymujemy tabelkę widoczną poniżej. Można tu zauważyć połączenia z konsumentem, jak i z producentem.

Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
[::1]:51811 ?	guest	■ running	○	AMQP 0-9-1	1	0 B/s	0 B/s
[::1]:51814 ?	guest	■ running	○	AMQP 0-9-1	1	759 KiB/s	0 B/s

Rysunek 3.17: Widoczne połączenia (connections) w RabbitMQ.

Po wybraniu "channels" otrzymujemy tabelkę widoczną poniżej. Wynika z tego, że kanał 51811 (połączenie z konsumentem) jest bezczynny, czyli nie wykonuje żadnych operacji i czeka. Z kolei kanał 51814 (związany z producentem) jest aktywny i wszystkie wiadomości, które wysyła są usuwane.

Overview				Details			Message rates				
Channel	User name	Mode ?	State	Unconfirmed	Prefetch ?	Unacked	publish	confirm	unroutable (drop)	deliver / get	ack
[::1]:51811 (1)	guest		■ idle	0		0					
[::1]:51814 (1)	guest		■ running	0		0	10,063/s	0.00/s	10,063/s		

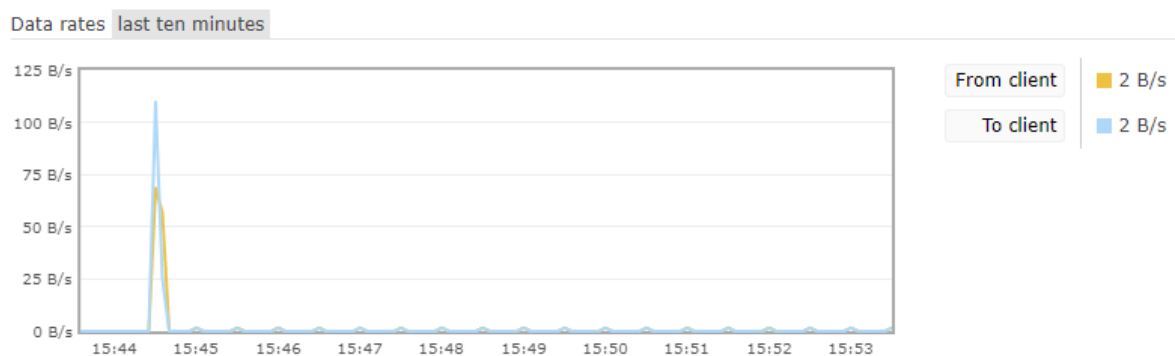
Rysunek 3.18: Widoczne kanały (channels) w RabbitMQ.

Po wybraniu "exchanges" otrzymujemy poniższą tabelkę. Wynika z niej, że wszystkie wymiany idą do *main exchange*.

Virtual host	Name	Type	Features	Message rate in	Message rate out
/	(AMQP default)	direct	D	0.00/s	0.00/s
/	alt_exchange	fanout			
/	altexchange	fanout			
/	amq.direct	direct	D		
/	amq.fanout	fanout	D		
/	amq.headers	headers	D		
/	amq.match	headers	D		
/	amq.rabbitmq.trace	topic	D I		
/	amq.topic	topic	D		
/	dead_letter_exchange	fanout			
/	main_exchange	direct		10,109/s	0.00/s
/	mainexchange	direct	AE	0.00/s	0.00/s

Rysunek 3.19: Widoczne wymiany (exchanges) w RabbitMQ.

Po wejściu w połączenie z konsumentem po paru minutach otrzymujemy poniższy wykres. Na początku połączenia jest widoczna większa aktywność, natomiast później widoczna jest bezczynność.



Rysunek 3.20: Wykres połączenia konsumenta z RabbitMQ.

Natomiast po wejściu w połączenie z producera po paru minutach otrzymujemy wykres widoczny poniżej. Na tym wykresie jest widoczny duży przesył wiadomości, który się utrzymuje na podobnym poziomie, a po jakimś czasie zmniejsza się.



Rysunek 3.21: Wykres połączenia producera z RabbitMQ.

3.2. Competing customers

Problem "competing consumers" w RabbitMQ dotyczy sytuacji, gdy wiele konsumentów konkuruje o przetwarzanie wiadomości z jednej kolejki. Może się to wiązać z pewnymi problemami. M.in. może dojść do zablokowania wiadomości albo jej zagubienia.

Competing customers ma również zalety. Najważniejsze z nich to:

- a) Wielozadaniowość: Każdy konsument może pełnić odrębną rolę lub przetwarzać różne typy wiadomości.
- b) Wydajność: Dzięki równoległemu przetwarzaniu przez wiele konsumentów, system RabbitMQ może efektywnie obsługiwać duże ilości wiadomości, dzięki czemu jest większa przepustowość.
- c) Niezawodność: W przypadku awarii jednego konsumenta, pozostali konsumenci nadal mogą kontynuować przetwarzanie wiadomości. To zapewnia większą niezawodność systemu.

Poniższy kod jest zaimplementowany dla konsumenta nr 1. Jego zadaniem jest implementacja konsumenta 1 w systemie RabbitMQ, który odbiera wiadomości z głównej kolejki o nazwie *main queue* i przetwarza je. Konsument 1 został skonfigurowany tak, że odbiera tylko te wiadomości, które są przypisane do niego za pomocą klucza routingu *consumer1*. Po odebraniu wiadomości, jej treść jest wyświetlana, a następnie konsument potwierdza odbiór wiadomości, co pozwala na jej usunięcie z kolejki.

```
1 # consumers competing - consumer 1
2
3 import pika
4
5 # Importowanie modułu pika, który jest biblioteką klienta do obsługi RabbitMQ.
6
7 # Nawiązywanie połączenia z RabbitMQ
8 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
9 channel = connection.channel()
10
11 # Deklarowanie głównej wymiany (exchange) i kolejki (queue)
12 main_exchange = 'main_exchange'
13 main_queue = 'main_queue'
14
15 # Deklarowanie głównej wymiany
16 channel.exchange_declare(exchange=main_exchange, exchange_type='direct')
17
18 # Deklarowanie głównej kolejki z opcjami
19 queue_arguments = {
20     'x-message-ttl': 10000, # Ustawienie czasu życia wiadomości (w milisekundach)
21     'x-dead-letter-exchange': 'dead_letter_exchange', # Ustawienie wymiany dla martwych wiadomości
22     'x-max-length': 100, # Ustawienie maksymalnej długości kolejki
23     'x-max-priority': 10 # Ustawienie maksymalnego priorytetu wiadomości
24 }
25 channel.queue_declare(queue=main_queue, durable=True, arguments=queue_arguments)
```

Rysunek 3.22: Konsument 1 - competing consumers, część 1.

```
27 # Powiązanie głównej kolejki z główną wymianą za pomocą klucza routingu "consumer1"
28 channel.queue_bind(exchange=main_exchange, queue=main_queue, routing_key='consumer1')
29
30 # Ustawienie funkcji zwrotnej (callback) dla konsumenta
31 def consumer_callback(ch, method, properties, body):
32     message = body.decode()
33     routing_key = method.routing_key
34     print(f"Otrzymano wiadomość: {message} z kluczem routingu: {routing_key}")
35
36     # Potwierdzenie odbioru wiadomości
37     ch.basic_ack(delivery_tag=method.delivery_tag)
38
39 # Rozpoczęcie odbierania wiadomości
40 channel.basic_consume(queue=main_queue, on_message_callback=consumer_callback)
41
42 print('Konsument 1 rozpoczął działanie. Oczekiwanie na wiadomości...')
43 channel.start_consuming()
```

Rysunek 3.23: Konsument 1 - competing consumers, część 2.

Poniższy kod jest zaimplementowany dla konsumenta nr 2. Jest on stworzony analogicznie do kodu dla konsumenta 1, tyle że ten odbiera wiadomości, które są przypisane do klucza routingu *consumer2*.

```
3 import pika
4
5 # Importowanie modułu pika, który jest biblioteka klienta do obsługi RabbitMQ.
6
7 # Nawiązywanie połączenia z RabbitMQ
8 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
9 channel = connection.channel()
10
11 # Deklarowanie głównej wymiany (exchange) i kolejki (queue)
12 main_exchange = 'main_exchange'
13 main_queue = 'main_queue'
14
15 # Deklarowanie głównej wymiany
16 channel.exchange_declare(exchange=main_exchange, exchange_type='direct')
17
18 # Deklarowanie głównej kolejki z opcjami
19 queue_arguments = {
20     'x-message-ttl': 10000, # Ustawienie czasu życia wiadomości (w milisekundach)
21     'x-dead-letter-exchange': 'dead_letter_exchange', # Ustawienie wymiany dla martwych wiadomości
22     'x-max-length': 100, # Ustawienie maksymalnej długości kolejki
23     'x-max-priority': 10 # Ustawienie maksymalnego priorytetu wiadomości
24 }
25 channel.queue_declare(queue=main_queue, durable=True, arguments=queue_arguments)
26
27 # Powiązanie głównej kolejki z główną wymianą
28 channel.queue_bind(exchange=main_exchange, queue=main_queue)
```

Rysunek 3.24: Konsument 2 - competing consumers, część 1.

```
30 # Ustawienie funkcji zwrotnej (callback) dla konsumenta
31 def consumer_callback(ch, method, properties, body):
32     message = body.decode()
33     routing_key = method.routing_key
34     print(f"Otrzymano wiadomość: {message} z kluczem routingu: {routing_key}")
35
36     # Potwierdzenie odbioru wiadomości
37     ch.basic_ack(delivery_tag=method.delivery_tag)
38
39     # Rozpoczęcie odbierania wiadomości
40     channel.basic_consume(queue=main_queue, on_message_callback=consumer_callback)
41
42     print('Konsument 2 rozpoczął działanie. Oczekiwanie na wiadomości...')
43     channel.start_consuming()
```

Rysunek 3.25: Konsument 1 - competing consumers, część 1.

Poniższy kod jest zaimplementowany dla producenta. Jest on odpowiedzialny za implementację producenta w systemie RabbitMQ, który publikuje wiadomości do kolejki. Wiadomości są wysyłane do głównej wymiany, a klucz routingu decyduje, którzy konsumenci odbiorą wiadomości.

```
1 import pika
2
3 # Importowanie modułu pika, który jest biblioteką klienta do obsługi RabbitMQ.
4
5 # Nawiązywanie połączenia z RabbitMQ
6 connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
7 channel = connection.channel()
8
9 # Deklarowanie głównej wymiany (exchange) i kolejki (queue)
10 main_exchange = 'main_exchange'
11 main_queue = 'main_queue'
12
13 # Deklarowanie głównej wymiany
14 channel.exchange_declare(exchange=main_exchange, exchange_type='direct')
15
16 # Deklarowanie głównej kolejki z opcjami
17 queue_arguments = {
18     'x-message-ttl': 10000, # Ustawienie czasu życia wiadomości (w milisekundach)
19     'x-dead-letter-exchange': 'dead_letter_exchange', # Ustawienie wymiany dla martwych wiadomości
20     'x-max-length': 100, # Ustawienie maksymalnej długości kolejki
21     'x-max-priority': 10 # Ustawienie maksymalnego priorytetu wiadomości
22 }
23 channel.queue_declare(queue=main_queue, durable=True, arguments=queue_arguments)
24
25 # Powiązanie głównej kolejki z główną wymianą
26 channel.queue_bind(exchange=main_exchange, queue=main_queue)
```

Rysunek 3.26: Producer - competing consumers, część 1.

```
28 # Publikowanie wiadomości
29 message_count = 1000000
30 for i in range(1, message_count + 1):
31     message = f'Wiadomość {i}'
32
33     if i % 2 == 0 or i % 3 == 0:
34         routing_key = 'consumer1'
35     else:
36         routing_key = 'consumer2'
37
38     channel.basic_publish(
39         exchange=main_exchange,
40         routing_key=routing_key,
41         body=message.encode(), # Kodowanie wiadomości jako bajty
42         properties=pika.BasicProperties(delivery_mode=2) # Ustawienie trwałości wiadomości
43     )
44     print(f'Opublikowano: {message} z kluczem routingu: {routing_key}')
45
46 # Zamknięcie połączenia
47 connection.close()
```

Rysunek 3.27: Producer - competing consumers, część 2

Po uruchomieniu powyższych kodów i wybraniu "connections" w RabbitMQ otrzymujemy tabelkę widoczną poniżej. Można tu zauważyć połączenia z konsumentami, jak i z producentem.

Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
[::1]:61910 ?	guest	■ running	○	AMQP 0-9-1	1	90 KiB/s	552 KiB/s
[::1]:61913 ?	guest	■ running	○	AMQP 0-9-1	1	86 KiB/s	526 KiB/s
[::1]:61916 ?	guest	■ running	○	AMQP 0-9-1	1	758 KiB/s	0 B/s

Rysunek 3.28: Widoczne połączenia (connections) w RabbitMQ.

Po wybraniu "exchanges" otrzymujemy poniższą tabelkę. Wynika z niej, że wszystkie wymiany idą do *main exchange*.

Exchanges

▼ All exchanges (12)

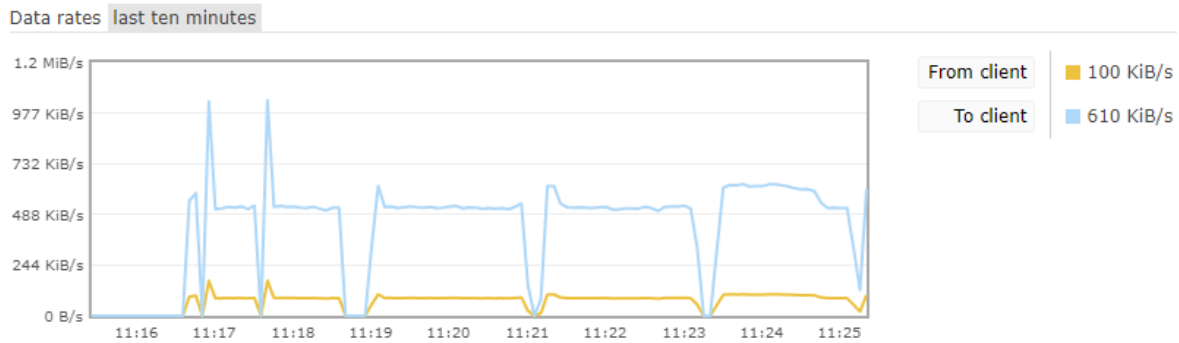
Pagination

Page 1 ▼ of 1 - Filter: ☐ Regex ?

Virtual host	Name	Type	Features	Message rate in	Message rate out
/	(AMQP default)	direct	D	0.00/s	0.00/s
/	alt_exchange	fanout			
/	altexchange	fanout			
/	amq.direct	direct	D		
/	amq.fanout	fanout	D		
/	amq.headers	headers	D		
/	amq.match	headers	D		
/	amq.rabbitmq.trace	topic	D I		
/	amq.topic	topic	D		
/	dead_letter_exchange	fanout			
/	main_exchange	direct		8,354/s	8,323/s
/	mainexchange	direct	AE	0.00/s	0.00/s

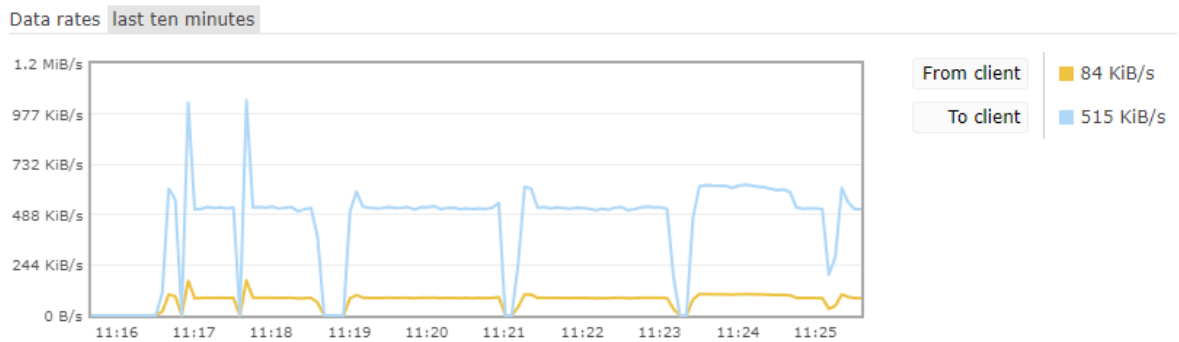
Rysunek 3.29: Widoczne wymiany (exchanges) w RabbitMQ.

Wykres połączenia konsumenta 1 wygląda następująco. Producent został włączony 4 razy.



Rysunek 3.30: Wykres połączenia konsumenta 1.

Natomiast wykres połączenia konsumenta 2 jest przedstawiony poniżej. Można zauważyć, że wykresy się nieznacznie różnią.



Rysunek 3.31: Wykres połączenia konsumenta 2.

Literatura

- [1] <http://weii.portal.prz.edu.pl/pl/materialy-do-pobrania>
- [2] <https://www.youtube.com/playlist?list=PLalrWAGybpB-UHbRDhFsBgXJM1g6T4IvO>
- [3] <https://www.rabbitmq.com/>
- [4] <https://www.rabbitmq.com/documentation.html>
- [5] <https://github.com/rabbitmq/rabbitmq-server>
- [6] <https://www.rabbitmq.com/dlx.html>
- [7] <https://www.rabbitmq.com/ae.html>