Systemy rozproszone i integracja usług - ćwiczenia nr 3: implementacja usługi JMS

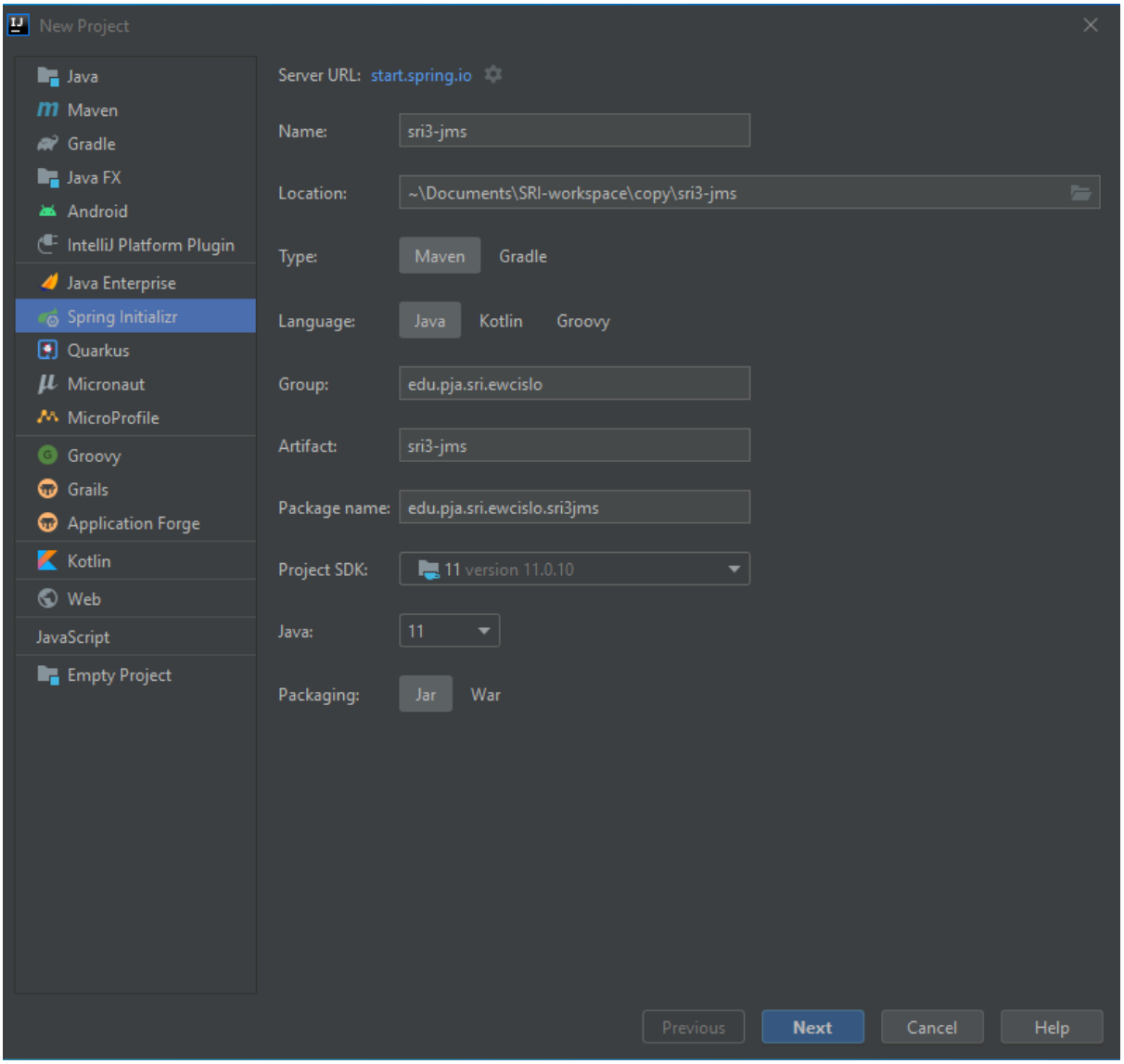
Celem ćwiczenia jest nabycie umiejętności implementacji usługi typu JMS w modelu point-to-point oraz publish-subscribe.

Część 1: zadanie wprowadzające

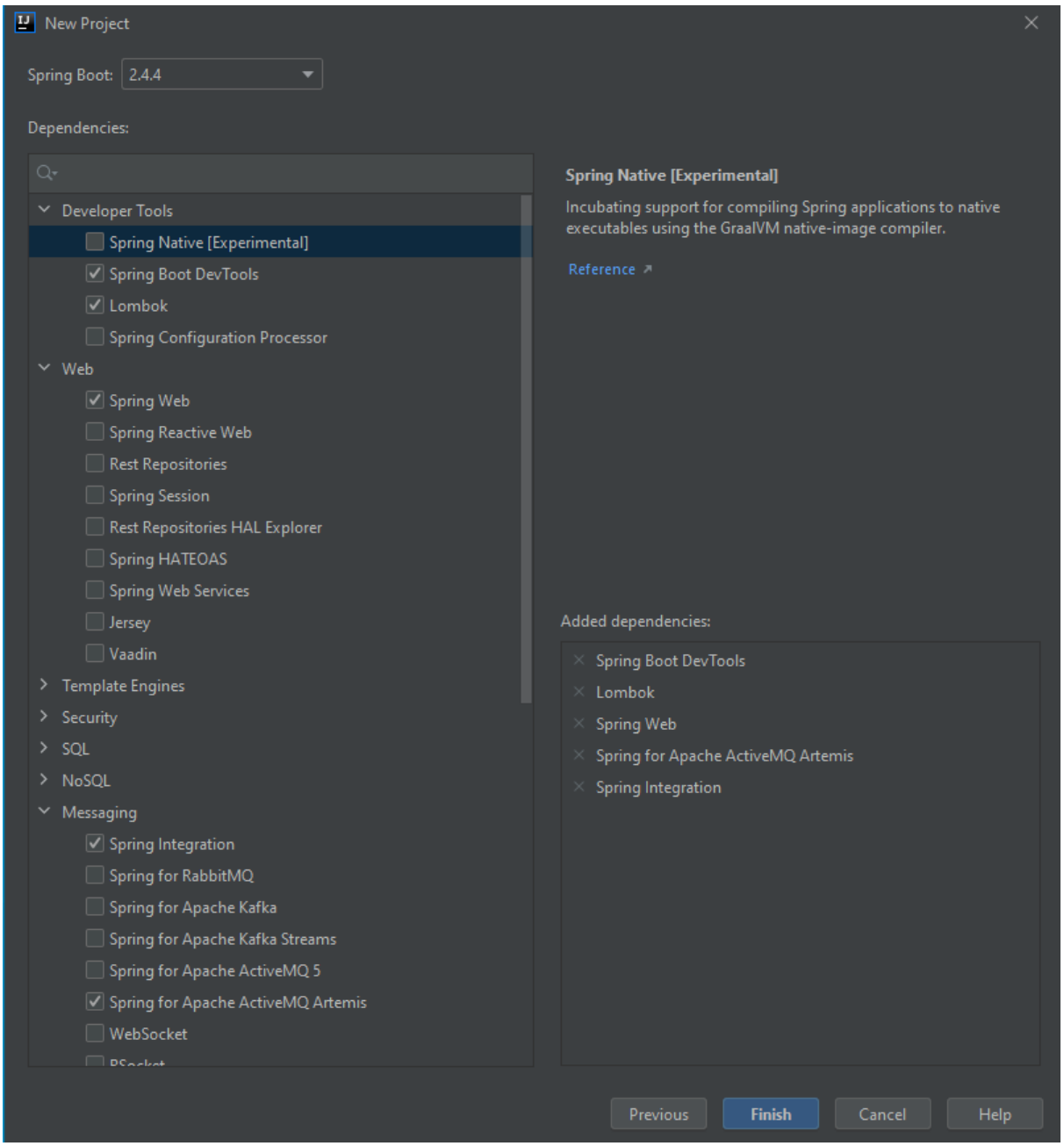
W ramach tego ćwiczenia zaimplementujemy przykładowe usługi JMS w technologii Java Spring. Wykorzystamy architekturę komunikatów typu point-to-point oraz publish-subscribe. Również przedstawiony zostanie przykład odpowiedzi na przychodzący komunikat JMS.

1. Niezbędne oprogramowanie.
   Do wykonania tego ćwiczenia niezbędne będzie środowisko Java (rekomendowane JDK Oracle 11 LTS), oraz środowisko developerskie (rekomendowane IntelliJ Idea Ultimate). Instrukcje instalacji znajdują się w opisie poprzedniego ćwiczenia.
2. Tworzenie szkieletu aplikacji
   Aplikacja, podobnie jak w poprzednim ćwiczeniu, będzie oparta o framework Spring Boot. W celu stworzenia szkieletu projektu należy wykorzystać narzędzie Spring Initializr (https://start.spring.io/). Projekt wykorzystujący powyższe narzędzie można również utworzyć w środowisku IntelliJ Idea Ultimate. W tym ćwiczeniu zaprezentujemy ten sposób.

2.1. Tworzenie nowego projektu
   W programie IntelliJ Idea Ultimate należy wybrać opcję New -> Project… z menu File Następnie należy wybrać ‘Spring Initializr’ z menu po lewej stronie okna dialogowego i uzupełnić podstawowe dane projektu:



Po wybraniu ‘Next’ należy zdefiniować zależności projektu, jak na poniższym zrzucie ekranu:



2.2. Uzupełnienie zależności w pom.xml

W tej aplikacji będziemy wykorzystywać broker JMS Apache ActiveMQ Artemis w trybie embedded, będzie on uruchamiany razem z aplikacją. W celu dołączenia powyższego middleware należy uzupełnić plik pom.xml o następujące zależności:

```
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>artemis-server</artifactId>
</dependency>
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>artemis-jms-server</artifactId>
</dependency>
```

Po zapisaniu zaktualizowanego pliku pom.xml należy kliknąć prawym klawiszem myszy na głównej gałęzi projektu w oknie po lewej stronie i z menu kontekstowego wybrać Maven -> Reload project. Nowe biblioteki powinny zostać pobrane i zainstalowane automatycznie.

3. Konfiguracja projektu
   Należy upewnić się, że JDK, procesor adnotacji i plugin Lombok zostały poprawnie skonfigurowane. Szczegółowy opis znajduje się w opisie poprzedniego zadania w punktach 3.1 - 3.3

3.1. Konfiguracja application.properties
W celu ustawienia brokera JMS należy dodać następujący fragment do pliku application.properties:

```
spring.artemis.mode=EMBEDDED
spring.artemis.host=localhost
spring.artemis.port=61616

spring.artemis.embedded.enabled=true

spring.jms.template.default-destination=my-queue-1

logging.level.org.apache.activemq.audit.base=WARN
logging.level.org.apache.activemq.audit.message=WARN
```

Ustawienia te pozwolą na uruchomienie brokera JMS w ramach procesu aplikacji, ograniczą też nieco liczbę logów brokera na konsoli.

4. Tworzenie modelu komunikatów.
   Architektura JMS umożliwia przesyłanie komunikatów różnego typu (https://www.ibm.com/docs/en/integration-bus/10.0?topic=structure-jms-message-types). W naszym przykładzie zastosujemy komunikaty tekstowe, w których przekazane będą dane obiektowe zapisane w formacie JSON. Takie rozwiązanie jest obecnie dość popularne, ma istotne przewagi nad komunikatami obiektowymi serializowanymi do ciągu bajtów (ObjectMessage). Pozwala na luźniejsze powiązania między aplikacjami, bez konieczności współdzielenia kodu modelu komunikatów. Komunikaty tekstowe mogą być również konsumowane przez odbiorców napisanych w innych językach niż Java. Stwórzmy zatem klasę HelloMessage reprezentującą strukturę danych przekazywaną w komunikacie:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.model;

import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import com.fasterxml.jackson.datatype.jsr310.deser.LocalDateTimeDeserializer;
import com.fasterxml.jackson.datatype.jsr310.ser.LocalDateTimeSerializer;
import lombok.*;

import java.time.LocalDateTime;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class HelloMessage {
```

```
    private static long idIndex = 0;
    public static long nextId() {
        return idIndex++;
    }

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonSerialize(using = LocalDateTimeSerializer.class)
    private LocalDateTime createdAt;
    private long id;
    private String message;

}
```

Właściwą treścią komunikatu będzie pole message typu String. Dodatkowo klasa uzupełniona jest polami pomocniczymi: id i createdAt, oraz metodą pomocniczą generującą unikalne id. Dla pola typu LocalDateTime należało zdefiniować za pomocą adnotacji mechanizm serializacji, aby był możliwy zapis i odczyt do formatu JSON. Możliwe jest rozbudowanie modelu komunikatu o bardziej złożoną strukturę danych.

5. Konfiguracja mechanizmu dziennika zadań.
   W naszym przykładzie komunikaty będą wysyłane automatycznie w określonych odstępach czasu. Aby to umożliwić, należy włączyć i skonfigurować mechanizm dziennika zadań. W tym celu należy utworzyć poniższą klasę konfiguracyjną:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.task.SimpleAsyncTaskExecutor;
import org.springframework.core.task.TaskExecutor;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.annotation.EnableScheduling;

@Configuration
@EnableScheduling
@EnableAsync
public class SchedulerConfig {

    @Bean
    TaskExecutor taskExecutor() {
        return new SimpleAsyncTaskExecutor();
    }
}
```

6. Konfiguracja usługi JMS
   W naszym przykładzie wykorzystamy trzy modele komunikacji: point-to-point (kolejki), publish-subscribe (tematy), oraz point-to-point z przekazaniem odpowiedzi zwrotnej. Najbardziej istotną różnicą pomiędzy kolejkami i tematami jest to, że komunikat wysłany na kolejkę zawsze będzie konsumowany przez jednego odbiorcę (nawet jeśli na danej kolejce nasłuchuje wielu odbiorców). Zarazem komunikat wysłany do tematu będzie odebrany przez wszystkich nasłuchujących na nim odbiorców. Domyślnie komunikacja jest asynchroniczna i jednokierunkowa, co ma bardzo korzystny wpływ na skalowalność aplikacji. W przykładzie z przekazaniem

odpowiedzi mechanizm JMS zastosuje tymczasową kolejkę zwrotną, przy pomocy której wysłana zostanie komunikat z odpowiedzią.
Dodatkowo należy określić metodę konwersji komunikatów tekstowych na obiekty. W tym celu należy utworzyć klasę konfiguracyjną JMSConfig:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.config;

import org.springframework.beans.factory.annotation.Qualifier;
import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactory
Configurer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;
import
org.springframework.jms.support.converter.MappingJackson2MessageConverter;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.MessageType;
import
org.springframework.jms.support.destination.DynamicDestinationResolver;

import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Session;

@Configuration
public class JmsConfig {
//...
```

W klasie tej na początek zdefiniujmy nazwy używanych kolejek i tematów:

```
public static final String QUEUE_HELLO_WORLD = "HELLO.QUEUE";
public static final String TOPIC_HELLO_WORLD = "HELLO.TOPIC";
public static final String QUEUE_SEND_AND_RECEIVE = "SEND_RECEIVE.QUEUE";
```

W nazwie istotny jest postfix (‘.QUEUE’ lub ‘.TOPIC’) za pomocą którego usługa JMS będzie odróżniać rodzaj komunikacji. Kolejka do odpowiedzi nie musi być definiowana, mechanizm JMS użyje do tego kolejki tymczasowej.

Aby umożliwić słuchaczom komunikatów połączenie do kanałów komunikacji odpowiedniego typu (kolejki i tematy) należy zdefiniować odpowiednie obiekty typu JmsListenerContainerFactory:

```
@Bean
public JmsListenerContainerFactory<?>
queueConnectionFactory(@Qualifier("jmsConnectionFactory") ConnectionFactory
connectionFactory, DefaultJmsListenerContainerFactoryConfigurer configurer) {
   DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
   configurer.configure(factory, connectionFactory);
   factory.setPubSubDomain(false);
   return factory;
```

```
}

@Bean
public JmsListenerContainerFactory<?>
topicConnectionFactory(@Qualifier("jmsConnectionFactory") ConnectionFactory
connectionFactory, DefaultJmsListenerContainerFactoryConfigurer configurer) {
   DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
   configurer.configure(factory, connectionFactory);
   factory.setPubSubDomain(true);
   return factory;
}
```

Istotną różnicą w konfiguracji obu tych obiektów jest własność ‘pubSubDomain’ ustalona z wartością false i true odpowiednio dla kolejek i tematów.

Nadawcy komunikatów będą używać zdefiniowanych wcześniej nazw destynacji. Aby mechanizm JMS mógł rozpoznać, czy destynacją jest kolejka czy temat, niezbędny jest obiekt pomocniczy wskazujący na typ destynacji na podstawie jego nazwy (a dokładniej na podstawie wspomnianego wyżej postfixu nazwy):

```
@Bean
public DynamicDestinationResolver destinationResolver() {
   return new DynamicDestinationResolver() {
       @Override
       public Destination resolveDestinationName(Session session, String
destinationName, boolean pubSubDomain) throws JMSException {
           if (destinationName.endsWith(".TOPIC")) {
               pubSubDomain = true;
           }
           return super.resolveDestinationName(session, destinationName,
pubSubDomain);
       }
   };
}
```

Kolejnym elementem konfiguracji jest mechanizm automatycznej konwersji obiektów na tekst w formacie JSON i vice versa:

```
@Bean
public MessageConverter messageConverter() {
   MappingJackson2MessageConverter converter = new
MappingJackson2MessageConverter();
   converter.setTargetType(MessageType.TEXT);
   converter.setTypeIdPropertyName("_type");
   return converter;
}
```

W nagłówku komunikatu w polu _type zostanie wskazana klasa modelu komunikatu. W przypadku, gdy odbiorca nie ma dostępu do klasy modelu, należy zastosować ręczną konwersję z formatu JSON na własny model danych. Nie będzie to istotne w naszym przykładzie, gdzie nadawca i odbiorca będą współdzielić kod klasy modelu komunikatu i konwersja odbędzie się automatycznie.

7. Tworzenie nadawcy komunikatów typu point-to-point

W tym punkcie utworzymy komponent, który będzie wysyłać komunikaty do kolejki w określonych odstępach czasu. W tym celu należy stworzyć nową klasę jak poniżej:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.producer;

import lombok.RequiredArgsConstructor;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import java.time.LocalDateTime;

@Component
@RequiredArgsConstructor
public class HelloWorldQueueProducer {

   private final JmsTemplate jmsTemplate;

   @Scheduled(fixedRate = 2000)
   public void sendHello() {
       HelloMessage message = HelloMessage.builder()
               .id(HelloMessage.nextId())
               .createdAt(LocalDateTime.now())
               .message("Hello world!")
               .build();
       jmsTemplate.convertAndSend(JmsConfig.QUEUE_HELLO_WORLD, message);
       System.out.println("HelloWorldQueueProducer.sendHello - sent message:
"+message);
   }
}
```

W celu wysłania komunikatu niezbędny będzie obiekt typu JmsTemplate dostarczony przez Spring. Aby oszczędzić nieco trudu tworzenia konstruktora inicjującego nasz komponent, zastosowano tutaj adnotację `@RequiredArgsConstructor` z biblioteki Lombok, z jej pomocą zostanie automatycznie utworzony konstruktor dla pola typu final.
Metoda sendHello() będzie wywoływana periodycznie co 2 sekundy przez mechanizm dziennika zadań, w tym celu oznaczono ją adnotacją `@Scheduled(fixedRate = 2000)`. W metodzie tworzony jest obiekt komunikatu za pomocą buildera (co jest nieco bardziej czytelną alternatywą dla konstruktorów). Aby skorzystać z buildera, należało wcześniej oznaczyć klasę modelu adnotacją @Builder z pakietu Lombok. Metoda convertAndSend obiektu wyśle pod wskazaną destynację (określoną przez pierwszy parametr) przekazany komunikat (parametr drugi). Nastąpi automatyczna konwersja obiektu komunikatu do formatu JSON i zostanie utworzony komunikat tekstowy, który zostanie przekazany do brokera komunikatów. Należy zwrócić uwagę, że dostarczenie komunikatu do brokera nie oznacza że w tym momencie wszyscy słuchacze otrzymali komunikat, będzie to wykonane asynchronicznie do wątku nadawcy komunikatu (w przeciwieństwie do synchronicznych wywołań np. przez REST).

Po uruchomieniu aplikacji na konsoli możemy zauważyć podobne logi:

```
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:19.513107700, id=0, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:21.509991, id=1, message=Hello world!)
```

```
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:23.523062900, id=2, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:25.514663600, id=3, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:27.521512500, id=4, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T13:55:29.513518900, id=5, message=Hello world!)
```

Oznaczają one, że usługa wysyłania działa poprawnie i co 2 sekundy wysyłany jest komunikat. Nie ma on natomiast odbiorców, których zdefiniujemy w kolejnych krokach.

8. Tworzenie odbiorców komunikatów typu point-to-point
   Należy utworzyć następujący komponent odbierający komunikaty:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.Message;

@Component
public class HelloWorldQueueReceiver1 {

   @JmsListener(destination = JmsConfig.QUEUE_HELLO_WORLD, containerFactory =
"queueConnectionFactory")
   public void receiveHelloMessage(@Payload HelloMessage convertedMessage,
                                   @Headers MessageHeaders messageHeaders,
                                   Message message) {
       System.out.println("HelloWorldQueueReceiver1.receiveHelloMessage, message:
"+convertedMessage);

   }
}
```

Metoda odbierająca komunikaty opisana została za pomocą adnotacji @JmsListener, która określa destynację na której nasłuchiwane są komunikaty. Parametr containerFactory wskazuje na komponent JmsListenerContainerFactory zdefiniowany w klasie JmsConfig. Po uruchomieniu aplikacji widzimy logi zawierające odebrany komunikat:

```
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:30:49.170723700, id=0, message=Hello world!)
HelloWorldQueueReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:30:49.170723700, id=0, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:30:51.178840800, id=1, message=Hello world!)
HelloWorldQueueReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:30:51.178840800, id=1, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:30:53.184641800, id=2, message=Hello world!)
HelloWorldQueueReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:30:53.184641800, id=2, message=Hello world!)
```

W celu zbadania zachowania kolejki komunikatów przy kilku odbiorcach, należy skopiować komponent odbierający komunikaty i nadać mu inną nazwę (np. HelloWorldQueueReceiver2):

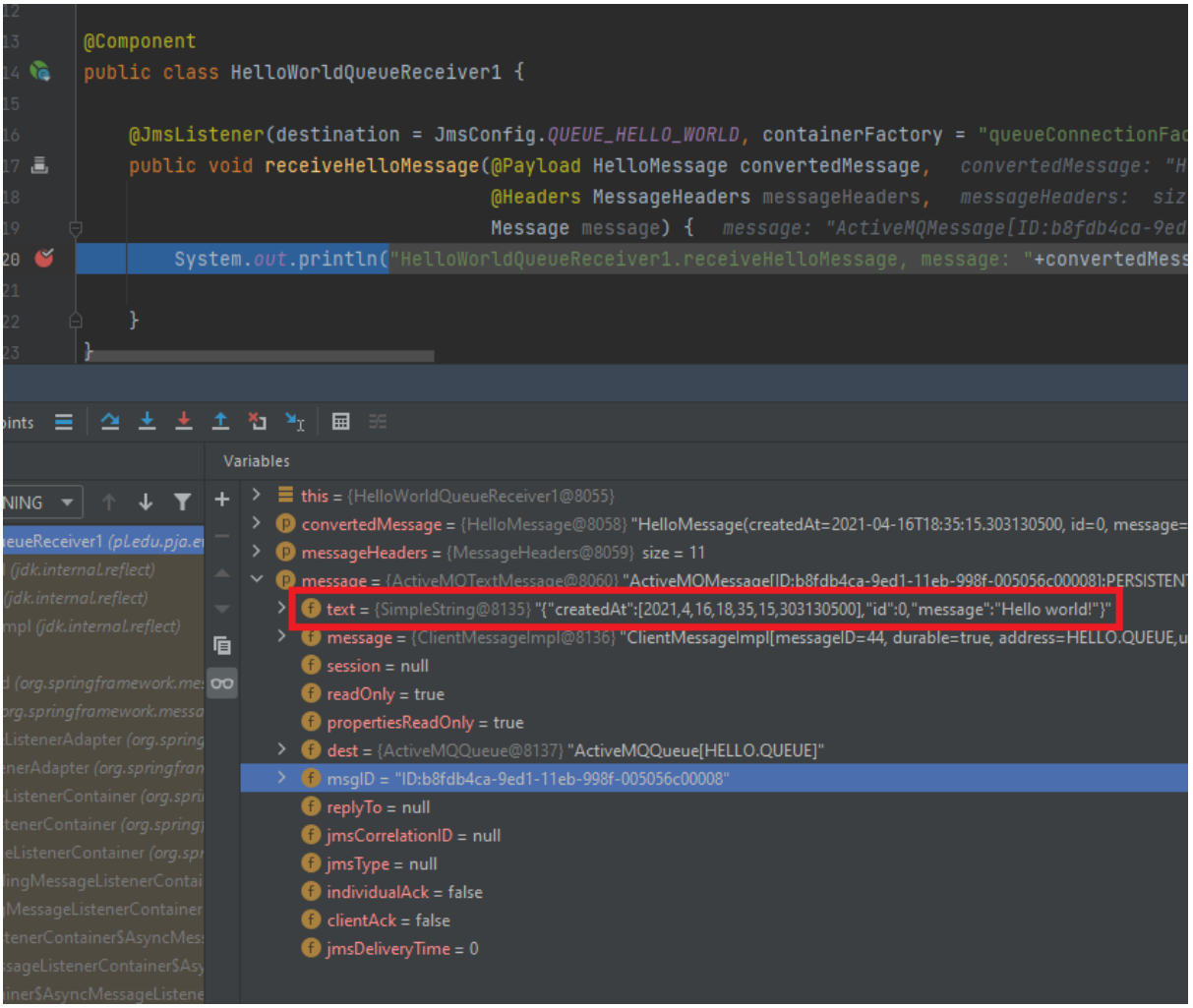
```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.Message;

@Component
public class HelloWorldQueueReceiver2 {

   @JmsListener(destination = JmsConfig.QUEUE_HELLO_WORLD, containerFactory =
"queueConnectionFactory")
   public void receiveHelloMessage(@Payload HelloMessage convertedMessage,
                                   @Headers MessageHeaders messageHeaders,
                                   Message message) {
       System.out.println("HelloWorldQueueReceiver2.receiveHelloMessage, message:
"+convertedMessage);
   }
}
```

Po uruchomieniu aplikacji widzimy następujące logi:

```
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:35:40.925601100, id=0, message=Hello world!)
HelloWorldQueueReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:35:40.925601100, id=0, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:35:42.934944300, id=1, message=Hello world!)
HelloWorldQueueReceiver2.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:35:42.934944300, id=1, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:35:44.934740800, id=2, message=Hello world!)
HelloWorldQueueReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:35:44.934740800, id=2, message=Hello world!)
HelloWorldQueueProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:35:46.931699800, id=3, message=Hello world!)
HelloWorldQueueReceiver2.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:35:46.931699800, id=3, message=Hello world!)
```

Widać zatem charakterystyczną cechę dla kolejek: dany komunikat jest przekazywany tylko do jednego odbiorcy.

Warto przeanalizować, jak wygląda komunikat przed konwersją na obiekt. W tym celu skorzystamy z debuggera i odczytamy własności obiektu Message przekazanego do metody. Należy ustawić breakpoint na linii kodu znajdującej się wewnątrz metody odbiorcy klikając dwukrotnie obok numeru linii w edytorze, a następnie uruchomić aplikację w trybie debug (domyślny skrót Shift-F9 w IntelliJ):



Po wstrzymaniu aplikacji mamy możliwość przeglądania zmiennych obecnych w danym kontekście wywołania. Można zobaczyć min. jak “fizycznie” wygląda treść komunikatu.

9. Tworzenie nadawcy komunikatów typu publish-subcribe
   Następnym krokiem jest przetestowanie mechanizmu wysyłania i odbioru komunikatów do tematu (topic). W tym celu należy utworzyć następujący komponent:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.producer;

import lombok.RequiredArgsConstructor;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import java.time.LocalDateTime;

@Component
@RequiredArgsConstructor
public class HelloWorldTopicProducer {

   private final JmsTemplate jmsTemplate;

   @Scheduled(fixedRate = 2500)
   public void sendHello() {
       HelloMessage message = HelloMessage.builder()
```

```
                .id(HelloMessage.nextId())
                .createdAt(LocalDateTime.now())
                .message("Hello world!")
                .build();
        jmsTemplate.convertAndSend(JmsConfig.TOPIC_HELLO_WORLD, message);
        System.out.println("HelloWorldTopicProducer.sendHello - sent message:
"+message);
    }
}
```

Główną różnicą w stosunku do nadawcy komunikatów na kolejkę jest inna nazwa destynacji. Wspomniany wcześniej mechanizm rozpoznawania typu destynacji po nazwie rozpozna ją jako temat zamiast kolejki.

10. Tworzenie odbiorców komunikatów typu publish-subscribe.
    Odbiorcami komunikatów z tematu będą następujące komponenty:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.Message;

@Component
public class HelloWorldTopicReceiver1 {

    @JmsListener(destination = JmsConfig.TOPIC_HELLO_WORLD, containerFactory =
"topicConnectionFactory")
    public void receiveHelloMessage(@Payload HelloMessage convertedMessage,
                                    @Headers MessageHeaders messageHeaders,
                                    Message message) {
        System.out.println("HelloWorldTopicReceiver1.receiveHelloMessage, message:
"+convertedMessage);

    }
}
```

---

oraz:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.receiver;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.Message;

@Component
public class HelloWorldTopicReceiver2 {
```

```
   @JmsListener(destination = JmsConfig.TOPIC_HELLO_WORLD, containerFactory =
"topicConnectionFactory")
   public void receiveHelloMessage(@Payload HelloMessage convertedMessage,
                                   @Headers MessageHeaders messageHeaders,
                                   Message message) {
       System.out.println("HelloWorldTopicReceiver2.receiveHelloMessage, message:
"+convertedMessage);
   }
}
```

Przed uruchomieniem programu warto wyłączyć logowanie z komponentów związanych z kolejkami, albo wyłączyć wysyłanie komunikatów na kolejkę poprzez wykomentowanie adnotacji @Scheduled w klasie HelloWorldQueueProducer.
Po uruchomieniu programu będą widoczne logi podobne do następujących:

```
HelloWorldTopicProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:50:08.459626100, id=0, message=Hello world!)
HelloWorldTopicReceiver2.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:08.459626100, id=0, message=Hello world!)
HelloWorldTopicReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:08.459626100, id=0, message=Hello world!)
HelloWorldTopicProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:50:10.966677800, id=1, message=Hello world!)
HelloWorldTopicReceiver2.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:10.966677800, id=1, message=Hello world!)
HelloWorldTopicReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:10.966677800, id=1, message=Hello world!)
HelloWorldTopicProducer.sendHello - sent message:
HelloMessage(createdAt=2021-04-16T17:50:13.468658100, id=2, message=Hello world!)
HelloWorldTopicReceiver1.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:13.468658100, id=2, message=Hello world!)
HelloWorldTopicReceiver2.receiveHelloMessage, message:
HelloMessage(createdAt=2021-04-16T17:50:13.468658100, id=2, message=Hello world!)
```

Widać, że w tym przypadku komunikat jest dystrybuowany do wszystkich odbiorców podłączonych pod dany temat.

11. Tworzenie nadawcy komunikatów typu point-to-point z odpowiedzią.
    W poprzednich scenariuszach komunikacja odbywała się w sposób jednokierunkowy - nadawca nie otrzymywał odpowiedzi, a nawet nie wiedział czy komunikat został poprawnie odczytany przez słuchaczy. W wielu scenariuszach jest dopuszczalne i pożądane - w ten sposób znacznie zwiększa się potencjał skalowalności aplikacji. W tym scenariuszu wypróbujemy mechanizm odpowiadania na komunikat, przesyłając komunikat zwrotny będący odpowiedzią dla nadawcy. Będzie to implementacja wzorca projektowego Request-reply. Należy utworzyć następujący komponent odpowiedzialny za wysyłanie pierwotnego komunikatu i odbiór odpowiedzi:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.producer;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.RequiredArgsConstructor;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.scheduling.annotation.Scheduled;
```

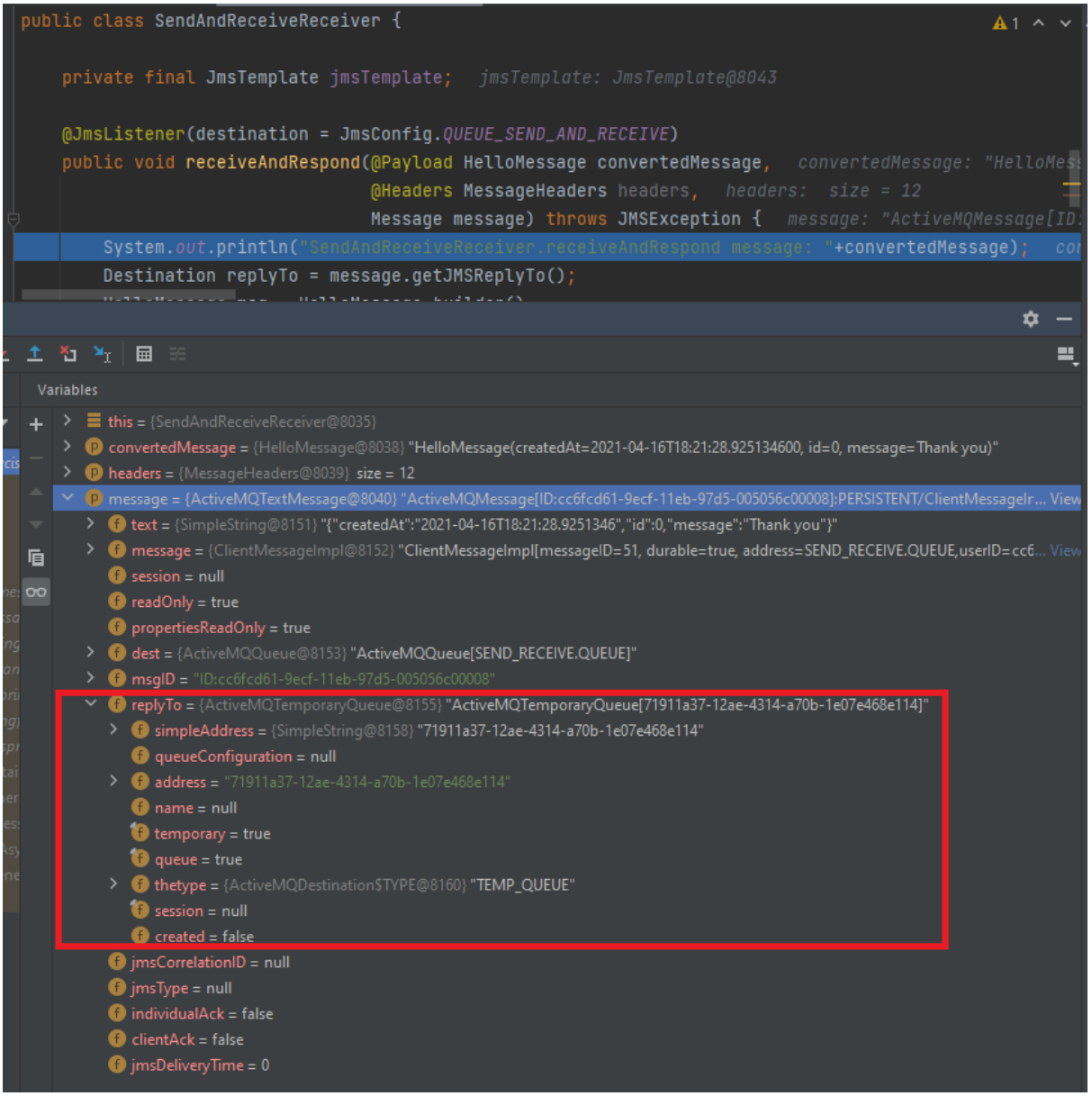
```
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import java.time.LocalDateTime;

@Component
@RequiredArgsConstructor
public class SendAndReceiveProducer {

   private final JmsTemplate jmsTemplate;
   private final ObjectMapper objectMapper;

   @Scheduled(fixedRate = 2000)
   public void sendAndReceive() throws JMSException, JsonProcessingException {
       HelloMessage message = HelloMessage.builder()
               .id(HelloMessage.nextId())
               .createdAt(LocalDateTime.now())
               .message("Thank you")
               .build();
       TextMessage responseMessage = (TextMessage) jmsTemplate.sendAndReceive(
               JmsConfig.QUEUE_SEND_AND_RECEIVE, new MessageCreator() {
           @Override
           public Message createMessage(Session session) throws JMSException {
               TextMessage plainMessage = session.createTextMessage();
               try {
                   plainMessage.setText(objectMapper.writeValueAsString(message));
                   plainMessage.setStringProperty("_type",
HelloMessage.class.getName());
                   return plainMessage;
               } catch (JsonProcessingException e) {
                   throw new JMSException("conversion to json failed: " +
e.getMessage());
               }
           }
       });
       String responseText = responseMessage.getText();

       HelloMessage responseConverted = objectMapper.readValue(responseText,
HelloMessage.class);
       System.out.println("SendAndReceiveProducer.sendAndReceive got response: "
               +responseText+"\n\tconvertedMessage: "+responseConverted);
   }
}
```

W obecnej wersji biblioteka Spring dla takiego scenariusza uniemożliwia automatyczną konwersję komunikatów z formatu JSON na obiekty Java i odwrotnie, zatem będzie konieczne użycie odpowiedniego mappera w kodzie komponentu. W tym celu podłączony został komponent `ObjectMapper` który używany jest podczas konwersji wysyłanego komunikatu z obiektu na tekstowy format JSON oraz podczas konwersji odebranego komunikatu z JSON na obiekt.
Komponent odbiorcy komunikatu wysyłającego odpowiedź będzie wyglądać następująco:

```
package pl.edu.pja.ewcislo.sri.jms.sri3jms.receiver;
```

```
import lombok.RequiredArgsConstructor;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.messaging.handler.annotation.Payload;
import org.springframework.stereotype.Component;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.config.JmsConfig;
import pl.edu.pja.ewcislo.sri.jms.sri3jms.model.HelloMessage;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import java.time.LocalDateTime;

@Component
@RequiredArgsConstructor
public class SendAndReceiveReceiver {

   private final JmsTemplate jmsTemplate;

   @JmsListener(destination = JmsConfig.QUEUE_SEND_AND_RECEIVE)
   public void receiveAndRespond(@Payload HelloMessage convertedMessage,
                                 @Headers MessageHeaders headers,
                                 Message message) throws JMSException {
        System.out.println("SendAndReceiveReceiver.receiveAndRespond message:
"+convertedMessage);
       Destination replyTo = message.getJMSReplyTo();
       HelloMessage msg = HelloMessage.builder()
               .id(HelloMessage.nextId())
               .createdAt(LocalDateTime.now())
               .message("You're welcome!")
               .build();
       jmsTemplate.convertAndSend(replyTo, msg);
   }
}
```

W nagłówku o nazwie jmsReplyTo pierwotnego komunikatu znajduje się informacja o destynacji na którą może być przesłana odpowiedź. Odbiorca komunikatu wysyła na ten adres komunikat zwrotny.
Warto przeanalizować, jakiego typu jest to destynacja, w tym celu należy debuggować aplikację ustawiając breakpoint na jednej z linii kodu metody odbiorcy komunikatu:



Jak widać, mechanizm JMS utworzył tymczasową kolejkę do transportu komunikatu zwrotnego.
Po uruchomieniu aplikacji (wskazane jest wcześniejsze usunięcie logowania z innych komponentów) widzimy logi o następującej treści:

```
SendAndReceiveReceiver.receiveAndRespond message:
HelloMessage(createdAt=2021-04-16T18:46:38.730171800, id=0, message=Thank you)
SendAndReceiveProducer.sendAndReceive got response:
{"createdAt":[2021,4,16,18,46,38,861996900],"id":1,"message":"You're welcome!"}
      convertedMessage: HelloMessage(createdAt=2021-04-16T18:46:38.861996900,
id=1, message=You're welcome!)
SendAndReceiveReceiver.receiveAndRespond message:
HelloMessage(createdAt=2021-04-16T18:46:40.733486500, id=2, message=Thank you)
SendAndReceiveProducer.sendAndReceive got response:
{"createdAt":[2021,4,16,18,46,40,738486300],"id":3,"message":"You're welcome!"}
      convertedMessage: HelloMessage(createdAt=2021-04-16T18:46:40.738486300,
id=3, message=You're welcome!)
```

Część 2: zadanie na ocenę

Zaprojektuj i zaimplementuj system wspomagający zespół uczestniczący w wyścigach Formuły 1, z użyciem komunikatów JMS, komponentów Spring, oraz kolejek i tematów komunikatów.

1. Aplikacja zainstalowana w bolidzie powinna wysyłać do systemu w pit-stopie bieżące informacje nt. pracy bolidu – temperaturę silnika, ciśnienie w oponach, ciśnienie oleju itp., oraz aktualny czas. Wysyłanie tych informacji powinno odbywać się automatycznie co 15 sekund, istotna jest szybkość działania i niezawodność. Aplikacja w bolidzie nie może ‘zawiesić się’ czekając na przetwarzanie przez odbiorcę wysłanych danych.

2. Komunikaty z bolidu powinny być przetwarzane przez dwóch niezależnych od siebie odbiorców. Zadaniem pierwszego z nich jest zapis przebiegu wyścigu, powinien on logować wszystkie przychodzące dane w czytelnej formie.

3. Drugim odbiorcą jest aplikacja monitorująca stan bolidu. W przypadku przekroczenia któregoś z parametrów jazdy powinna ona niezwłocznie powiadomić kierowcę o zaistniałej sytuacji. W przypadku poważnej awarii należy również powiadomić mechaników, aby byli przygotowani na naprawę bolidu w pit-stopie (należy skorzystać z wzorca projektowego Message Router)

4. Kierowca bolidu może zgłosić potrzebę zjazdu do pit-stopu. Kierownik zespołu rozważa jego prośbę i akceptuje ją, lub odrzuca. Kierowca zostanie powiadomiony o tej decyzji (należy skorzystać z wzorca Request-reply). Wysłanie komunikatu do kierownika i odbiór odpowiedzi powinien być w jednej metodzie.

Całość może zostać zaimplementowana jako pojedyncza aplikacja, której poszczególne komponenty komunikują się za pomocą JMS.