Designing a Message Channel Strategy for a Messaging Solution

Steven Haines
PRINCIPAL SOFTWARE ARCHITECT

@geekcap www.geekcap.com

Overview

Message channels

Point-to-point and publish-subscribe channels

Pollable vs. subscribable channels

Spring Integration channel implementations

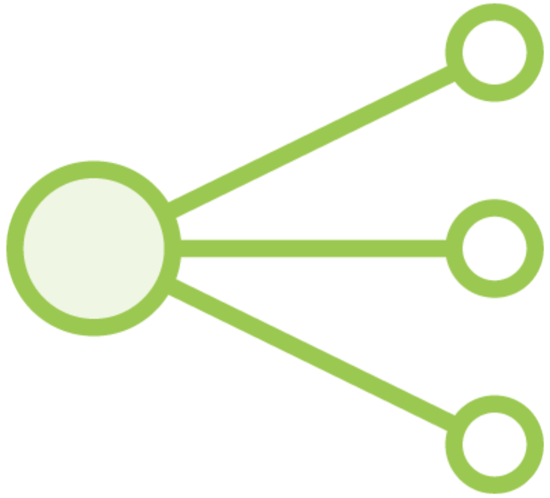
Message Channel

When two applications wish to exchange data, they do so by sending the data through a channel that connects the two.

Message Channel Semantics

Point-to-Point

A point-to-point semantic states that no more than one consumer can receive each message



Publish-subscribe

A publish-subscribe semantic attempts to broadcast each message to all subscribers

Message Buffering

Pollable Channel

Can buffer messages

Subscribable Channel

Cannot buffer messages

Spring Integration Channel Implementations

Publish Subscribe Channel

Queue Channel

Priority Channel

Rendezvous Channel

Direct Channel

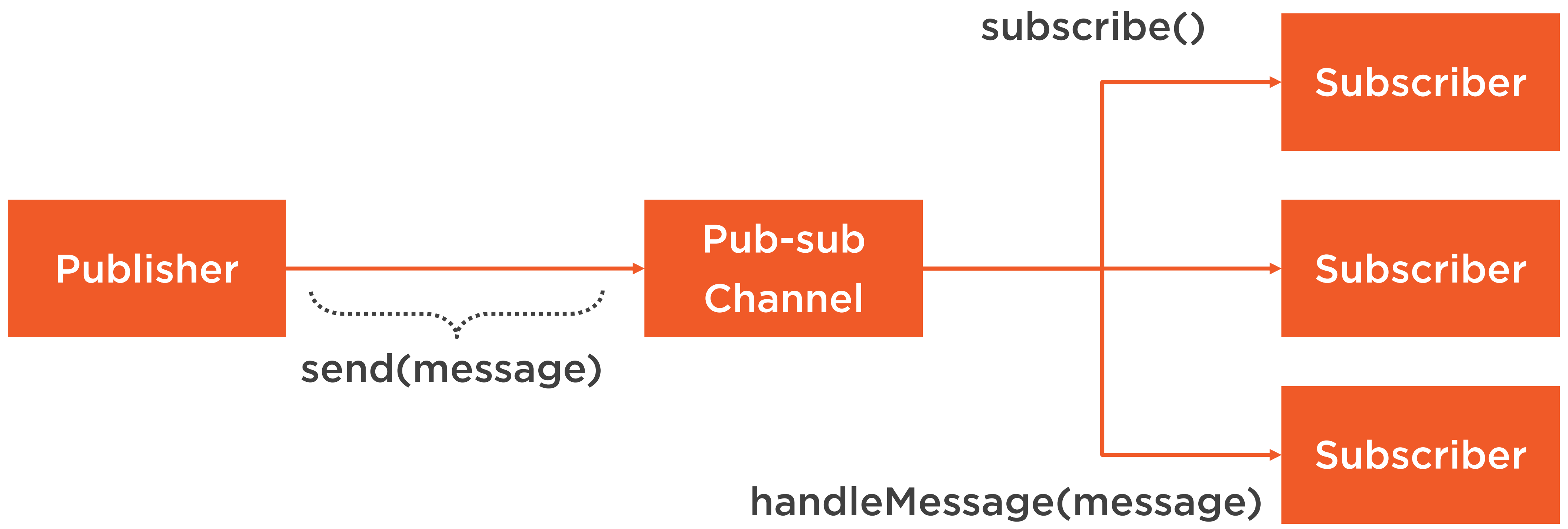
Executor Channel

Publish-subscribe Channel

Publish-subscribe Channel

The PublishSubscribeChannel is a subscribable channel that provides a publish-subscribe semantic that broadcasts messages sent to it to all its subscribed handlers.

How the Publish-subscribe Channel Works



As a subscribable channel, the PublishSubscribeChannel does not buffer messages
If the subscriber is not present when the message is published,
it does not receive the message

```
@Configuration
@EnableIntegration
public class EventMessagePatternConfig {
    @Bean
    public MessageChannel eventChannel() {
        return new PublishSubscribeChannel();
    }
}


@MessagingGateway(name = "eventGateway",
                  defaultRequestChannel = "eventChannel")
public interface EventGateway {
    @Gateway
    void publishEvent(Message<Event> event);
}


@Service
public class RegistrationServiceImpl {
    @Autowired
    private EventGateway eventGateway;
    public void notifyObservers(Event event) {
        Message<Event> message = MessageBuilder.withPayload(event)
        .setHeader(IntegrationMessageHeaderAccessor.EXPIRATION_DATE,
                System.currentTimeMillis() + 60 * 60 * 1000)
                .build();
        eventGateway.publishEvent(message);
    }
}


@Service
public class EventListenerOne {
    @ServiceActivator(inputChannel = "eventChannel")
    public void receivedEvent(Message<Event> message) {
        logger.info("EventListenerOne::received event: {}",
                message.getPayload());
    }
}
```

◂ Define a channel (Publish/Subscribe)

◂ Define a Gateway

◂ Publish to the Gateway

◂ Handle the message

Summary

A publish-subscribe channel is a subscribable channel that publishes messages to all of its subscribers

It is best used for Event Messages

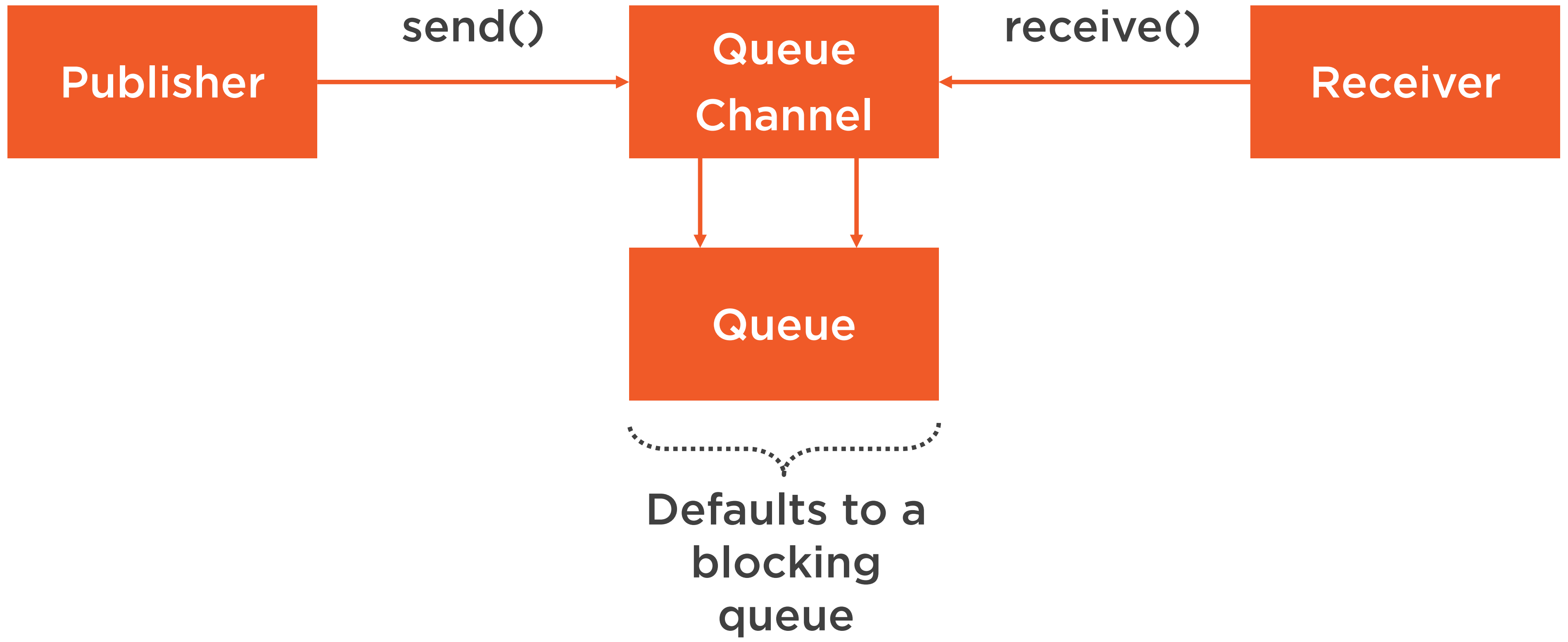
Next up: Queue Channel

Queue Channel

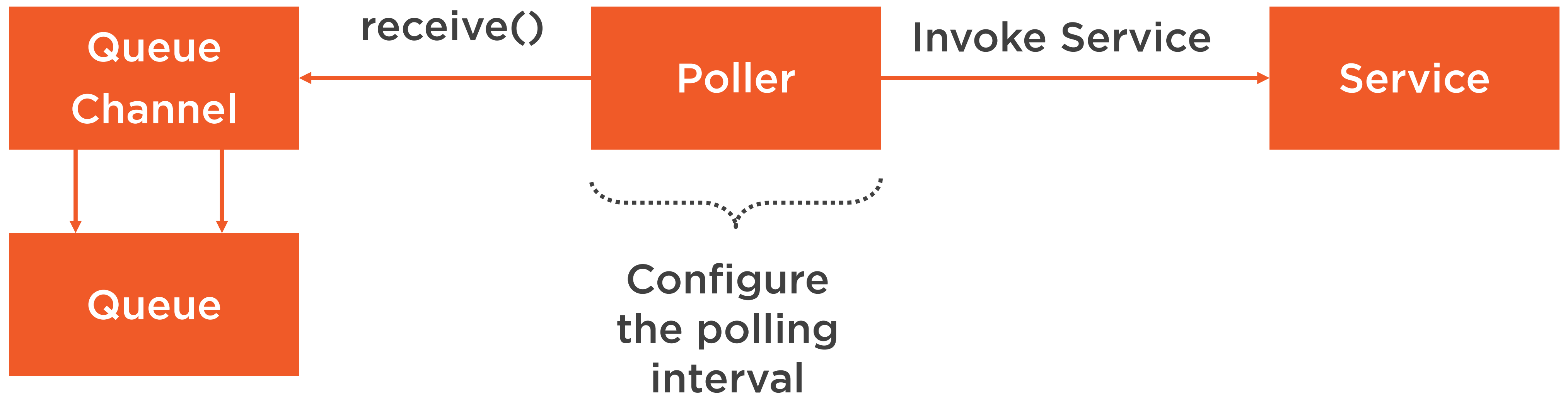
Queue Channel

A Queue Channel is a pollable channel that provides a point-to-point semantic by storing its messages in a queue and returning messages to receivers through its receive() method.

How the Queue Channel Works



Pollers and Service Activators



Poller Triggers

Trigger	Description
fixedDelay	A delay, in milliseconds, after a message is processed and the channel is polled
fixedRate	A rate, in milliseconds, in which the channel is polled
cron	A cron time configuration that defines when the channel is polled

```
@Configuration
@EnableIntegration
public class QueueChannelConfig {
    @Bean
    public MessageChannel queueChannel() {
        return new QueueChannel(10);
    }
}
```

◀ Define a queue channel – queue size of 10 in this example

```
@MessagingGateway(name = "queueChannelGateway",
        defaultRequestChannel = "queueChannel")
public interface QueueChannelGateway {
    @Gateway
    void sendSwag(Message<Swag> swag);
}
```

◀ Define a Gateway

```
@Service
public class RegistrationServiceImpl {
    @Autowired
    private QueueChannelGateway queuedSwagGateway;
    public void commit(String userId) {
        queuedSwagGateway.sendSwag(
            MessageBuilder.withPayload(
                        new Swag("T-Shirt")).build());
    }
}
```

◀ Publish to the Gateway

```
@Service
public class QueuedSwagServiceImpl {
    @ServiceActivator(inputChannel = "queueChannel",
                        poller = @Poller(fixedDelay = "100"))
    public void sendSwag(Message<Swag> swag) {
        logger.info("Received message to send swag: {}",
                    swag.getPayload());
    }
}
```

◀ Handle the message – the poller will poll the queue channel every 100ms

Demo

Define our components

- Queue Channel
- Queue Gateway
- Registration Service
- Queueable Swag Service with service activator

Invoke the Registration Service

Publish 4 messages using the Queue Gateway

Observe each message being processed one-at-a-time

Add Spring Application shutdown code to stop the poller thread

Summary

A queue channel is a pollable channel backed by a queue

Allows consumers to throttle incoming messages

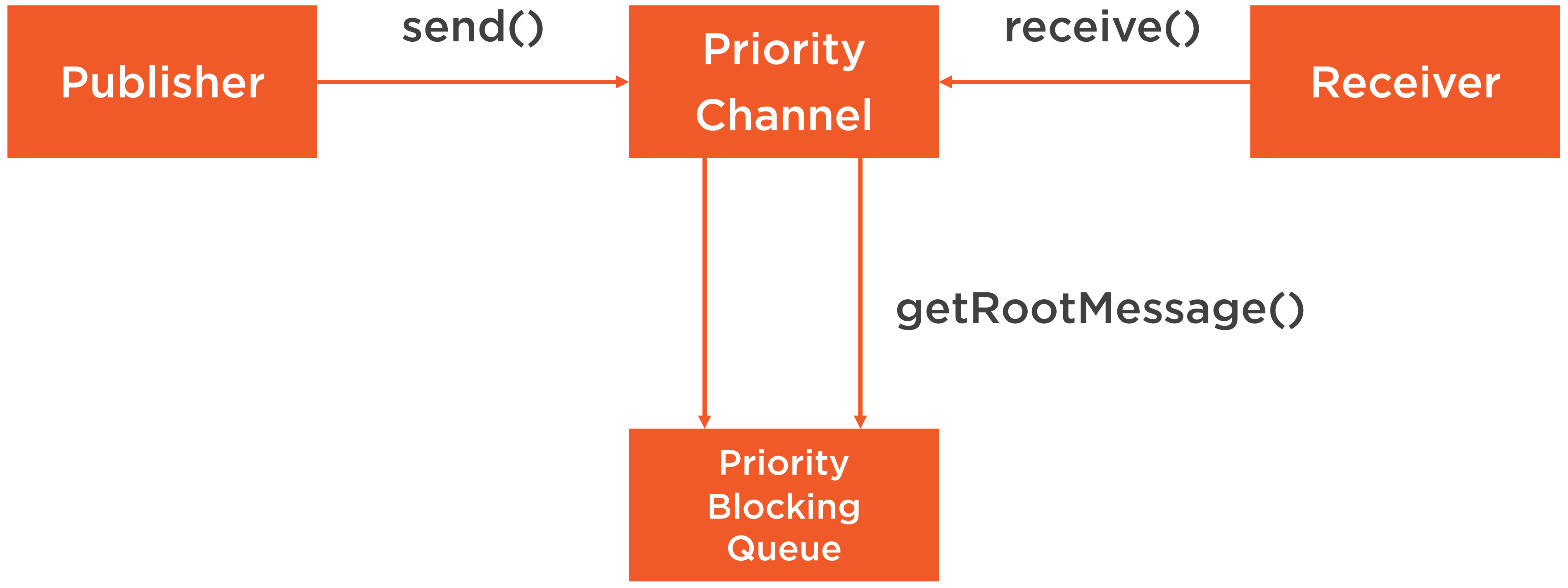
Next up: Priority Channel

Priority Channel

Priority Channel

A Priority Channel is a pollable and buffered channel that allows for messages to be ordered within the channel based upon a priority

How the Priority Channel Works



```
MessageBuilder
   .withPayload(payload)
   .setHeader(IntegrationMessageHeaderAccessor.PRIORITY,
              priority)

   .build();
```

Message Prioritization

By default, messages are prioritized using their PRIORITY message header

Priority is an Integer and messages with the highest priority are handled first

```
new PriorityChannel(
        Comparator.comparingLong((Message<?> m) ->
            ((Swag)m.getPayload())
                .getAmount())
                .reversed())
```

Custom Message Prioritization

A Comparator can be passed to the PriorityChannel's constructor to define custom message ordering

In this example we are prioritizing Swag objects by their amount (cost), descending, so the most expensive items will be processed first

Why Use Priority Channels?

Benefits

Allows you to control the order in which messages are processed

Higher priority messages will always be processed before lower priority messages

Drawbacks

Potential to starve messages with low priority

```
@Configuration
@EnableIntegration
public class PriorityChannelConfig {
    @Bean
    public MessageChannel priorityChannel() {
        return new PriorityChannel();
    }
    @Bean
    public MessageChannel customizedPriorityChannel() {
        return new PriorityChannel(Comparator.comparingLong(
                   (Message<?> m) ->
                    ((Swag)m.getPayload()).getAmount()).reversed());
    }
}

@MessagingGateway(name = "priorityChannelGateway",
        defaultRequestChannel = "priorityChannel")
public interface PriorityChannelGateway {
    @Gateway
    void sendSwag(Message<Swag> swag);
}
@MessagingGateway(name = “customizedPriorityChannelGateway",
        defaultRequestChannel = “customizedPriorityChannel")
public interface CustomizedPriorityChannelGateway {
    @Gateway
    void sendSwag(Message<Swag> swag);
}

@Service
public class RegistrationServiceImpl {
    @Autowired
    private PriorityChannelGateway swagGateway;
    public void commit(String userId) {
        swagGateway.sendSwag(MessageBuilder
            .withPayload(new Swag("T-Shirt", 5))
            .setHeader(IntegrationMessageHeaderAccessor.PRIORITY,
                       10).build());
    }
}
```

- Define two priority channels, one that prioritizes by priority and one that prioritizes by Swag amount
- Define Gateway for the priority channel
- Define a Gateway for the customized priority channel
- Publish messages to the Gateways

Demo

Define our components

- Priority Channels
- Priority Gateways
- Registration Service
- Swag Service with service activator

Invoke the Registration Service

Publish 5 messages using the Priority Gateways with different priorities and amounts

Observe the order in which each message is processed

Summary

A priority channel is a pollable channel backed by a priority blocking queue

Messages are prioritized based on their PRIORITY message header or by a custom prioritization comparator

Ensures that higher priority messages are processed first, but need to be cautious that low priority messages are not ignored

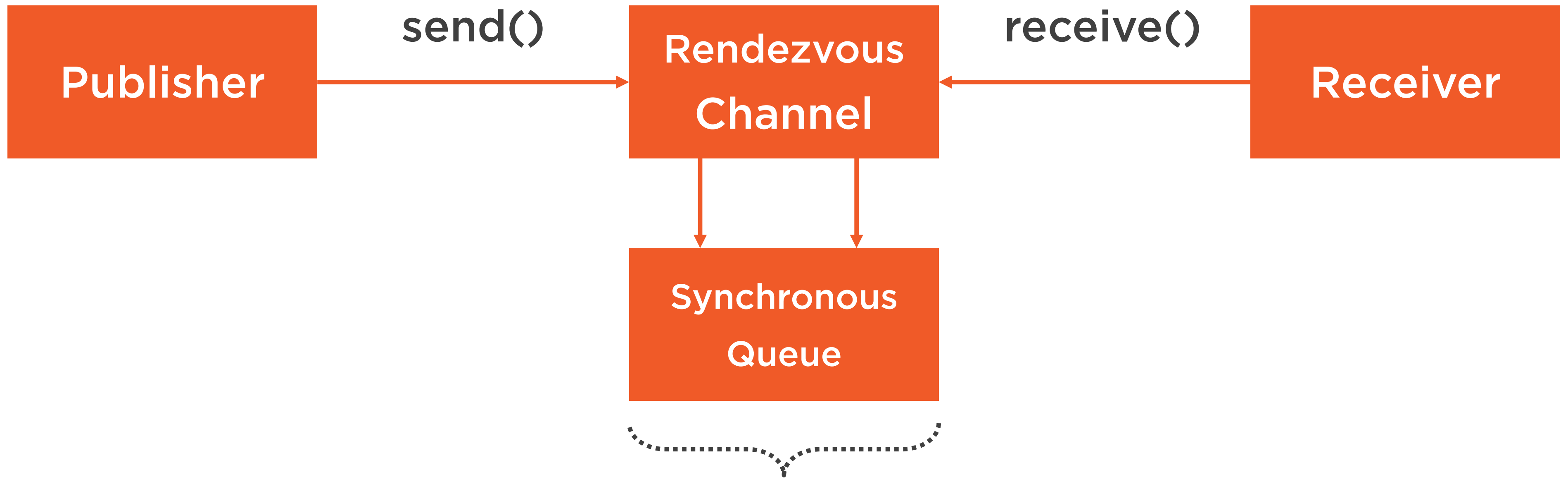
Next up: Rendezvous Channel

Rendezvous Channel

Rendezvous Channel

The Rendezvous Channel is a pollable point-to-point channel that enables a “direct-handoff” scenario, wherein a sender blocks until another party invokes the channel’s receive() method. The other party blocks until the sender sends the message.

How the Rendezvous Channel Works



Only holds a single message and blocks until that message is processed

```
public class RendezvousChannel extends QueueChannel {

    public RendezvousChannel() {

        super(new SynchronousQueue<Message<?>>());

    }

}
```

A Queue Channel with a Synchronous Queue

Everything you have learned about the Queue Channel applies to the Rendezvous Channel, but with a synchronous queue

A Synchronous Queue is a blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

When to Use the Rendezvous Channel

Point-to-Point Channel
Communication from one component to another component

Pollable
Requires that you configure a poller to receive messages

Request-Reply Messages
This is the best channel for the Request-Reply Message Pattern

```
@Configuration
@EnableIntegration
public class RendezvousChannelConfig {
    @Bean
    public MessageChannel rendezvousChannel() {
        return new RendezvousChannel();
    }
}


@MessagingGateway(name = "rendezvousChannelGateway",
                  defaultRequestChannel = "rendezvousChannel")
public interface RendezvousChannelGateway {
    @Gateway
    Message<Boolean> checkHotelAvailability(Message<Integer> partySize);
}


@Service
public class RegistrationServiceImpl {
    @Autowired
    private RendezvousChannelGateway gateway;
    public boolean checkHotelAvailability(int partySize) {
        Message<Boolean> available = gateway.checkHotelAvailability(
                      MessageBuilder.withPayload(partySize).build());
        return available.getPayload();
    }
}


@Service
public class HotelBookingServiceImpl implements HotelBookingService {
    @ServiceActivator(inputChannel = "rendezvousChannel",
                        poller = @Poller(fixedDelay = "100"))
    public Message<Boolean> checkAvailability(Message<Integer> size) {
        int numberOfGuests = partySize.getPayload();
        return numberOfGuests < 10 ?
               MessageBuilder.withPayload(true).build() :
               MessageBuilder.withPayload(false).build();
    }
}
```

◂ Define a Rendezvous Channel

◂ Define a Gateway

◂ Publish to the Gateway

◂ Handle the message – if the party size is under 10 return true, else return false

Demo

Define our components

- Rendezvous Channel
- Rendezvous Gateway
- Registration Service
- Hotel Booking Service with service activator

Invoke the Registration Service with options to get a positive and negative response

Observe that the message calls block in the rendezvous channel

Summary

A rendezvous channel is a pollable channel backed by a synchronous queue

Senders block on the send() method until the receiver responds

Best choice for implementing the Request-Reply Message Pattern

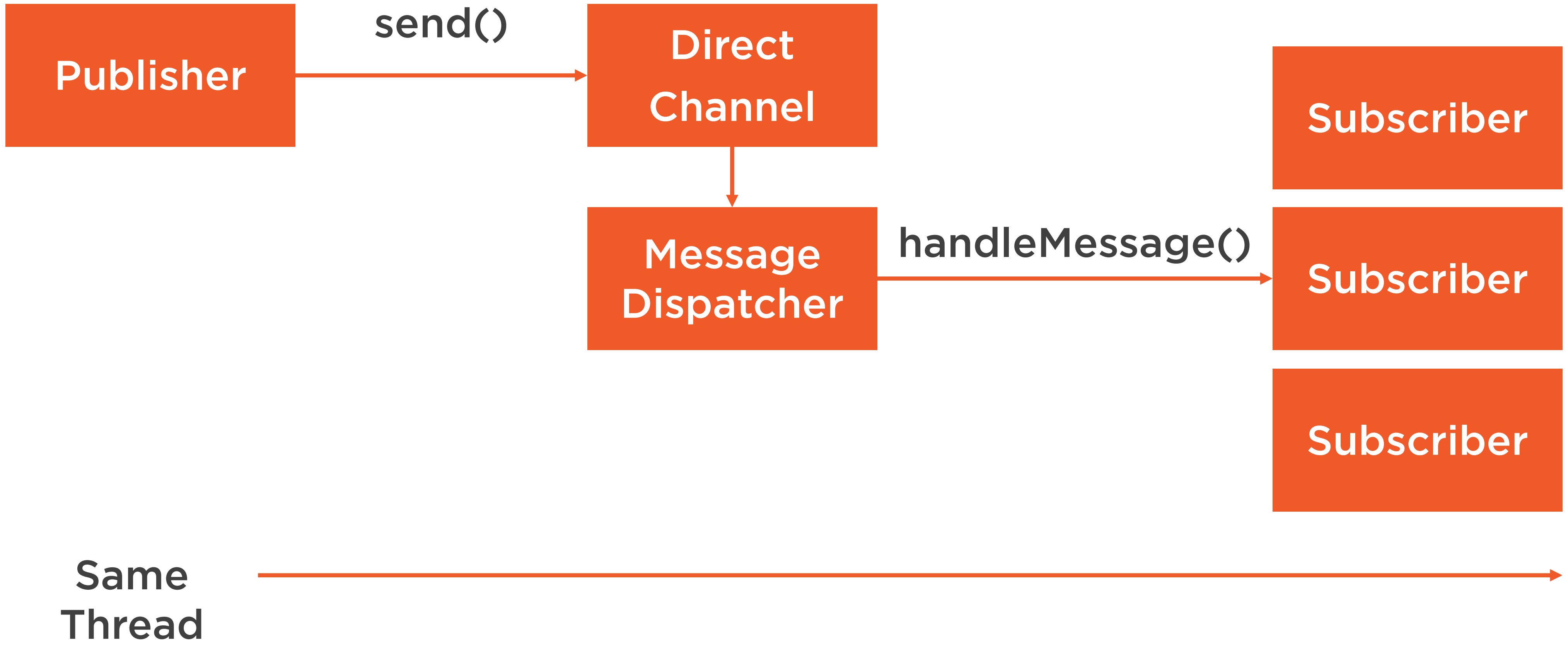
Next up: Direct Channel

Direct Channel

Direct Channel

The Direct Channel is a subscribable point-to-point channel that enables a single thread to perform operations on both sides of the channel.

How the Direct Channel Works



Motivation: Transaction Integrity

The key motivation for providing a channel implementation with this behavior is to support transactions that must span across the channel while still benefiting from the abstraction and loose coupling that the channel provides.

When to Use the Direct Channel

Point-to-Point Channel
Communication from one component to another component

Subscribable
Does not require that you configure a poller to receive messages, messages are sent on demand

Transactional
Supports transactions that span across channel communications

```
@Configuration
@EnableIntegration
public class DirectChannelConfig {
    @Bean
    public MessageChannel directChannel() {
        return new DirectChannel();
    }
}

@MessagingGateway(name = "directChannelGateway",
        defaultRequestChannel = "directChannel")
public interface DirectChannelGateway {
    @Gateway
    void createReservationRecord(Message<String> lastName);
}

@Service
public class RegistrationServiceImpl {
    @Autowired
    private DirectChannelGateway directChannelGateway;
    public void setupReservation(String ... lastNames) {
        for (String lastName :lastNames ) {
            directChannelGateway.createReservationRecord(
                      MessageBuilder.withPayload(lastName).build());
        }
    }
}

@Service
public class ReservationRecordServiceImpl implements ReservationRecordService {
    @ServiceActivator(inputChannel = "directChannel")
    public void createReservationRecord(Message<String> lastName) {
        logger.info("Creating reservation record for user {}",
                lastName.getPayload());
    }
}
```

◂ Define a Direct Channel

◂ Define a Gateway

◂ Publish to the Gateway

◂ Handle the Message

Demo

Define our components

- Direct Channel
- Direct Gateway
- Registration Service
- Registration Record Service with service activator

Invoke the Registration Service

Observe that the message calls are synchronous and use the same thread

Summary

A direct channel is a subscribable point-to-point channel

Senders block on the send() method until the receiver completes processing the message

Best choice for transactional calls that span across a channel, while retaining loose coupling

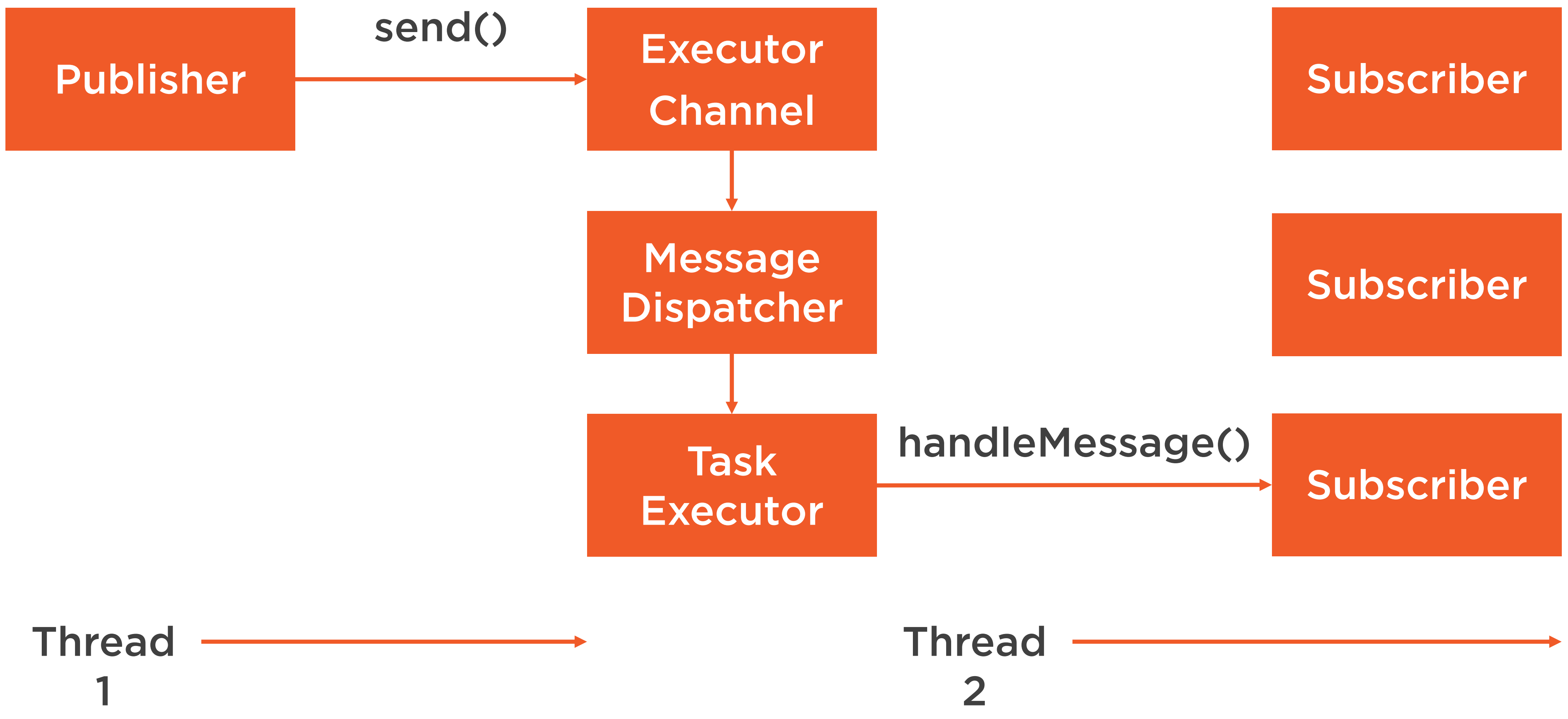
Next up: Executor Channel

Executor Channel

Executor Channel

The Executor Channel is a subscribable point-to-point channel that delegates to an instance of TaskExecutor to perform message dispatching.

How the Executor Channel Works



When to Use the Executor Channel

Point-to-Point Channel
Communication from one component to another component

Subscribable
Does not requires that you configure a poller to receive messages, messages are sent on demand

Multi-threaded
Supports execution by multiple threads in a Task Executor

```
@Configuration
@EnableIntegration
public class ExecutorChannelConfig {
   @Bean
   public MessageChannel executorChannel(TaskExecutor taskExecutor) {
      return new ExecutorChannel(taskExecutor);
   }
}
```

◀ Define an Executor Channel

```
@MessagingGateway(name = "executorChannelGateway",
      defaultRequestChannel = "executorChannel")
public interface ExecutorChannelGateway {
   @Gateway
   void createReservationRecord(Message<String> lastName);
}
```

◀ Define a Gateway

```
@Service
public class RegistrationServiceImpl {
   @Autowired
   private ExecutorChannelGateway executorChannelGateway;
   public void setupReservation(String ... lastNames) {
      for (String lastName :lastNames ) {
         executorChannelGateway.createReservationRecord(
                   MessageBuilder.withPayload(lastName).build());
      }
   }
}
```

◀ Publish to the Gateway

```
@Service
public class ReservationRecordServiceImpl implements ReservationRecordService {
   @ServiceActivator(inputChannel = “executorChannel")
   public void createReservationRecord(Message<String> lastName) {
      logger.info("Creating reservation record for user {}",
               lastName.getPayload());
   }
}
```

◀ Handle the Message

Demo

Define our components

- Executor Channel
- Executor Gateway
- Registration Service
- Registration Record Service with service activator

Invoke the Registration Service

Observe that the message calls are executed in different threads

Summary

An executor channel is a subscribable point-to-point channel

Message handling is performed by different threads controlled by a Task Executor

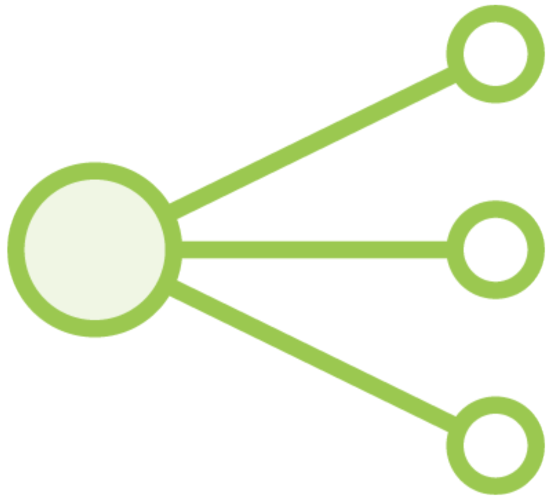
Best choice for non-blocking multi-threaded messages

Conclusion

Message Channel Semantics

Point-to-Point

A point-to-point semantic states that no more than one consumer can receive each message



Publish-subscribe

A publish-subscribe semantic attempts to broadcast each message to all subscribers

Message Buffering

Pollable Channel

Can buffer messages

Subscribable Channel

Cannot buffer messages

Spring Integration Channel Implementations

Publish Subscribe Channel

Queue Channel

Priority Channel

Rendezvous Channel

Direct Channel

Executor Channel

Summary

You should now understand how the core message channels work in Spring Integration

You should feel comfortable with publish-subscribe and point-to-point semantics, as well as pollable vs. subscribable implementations

You should be able to choose the right channel type for your business needs