Designing an Error Handling Strategy for Different Integration Scenarios

Steven Haines
PRINCIPAL SOFTWARE ARCHITECT

@geekcap www.geekcap.com

Overview

- Challenges in handling errors in an asynchronous application
- Handling errors in request-reply channels
- Handling errors in asynchronous channels
- Error Channels
- Dead Letter Channels

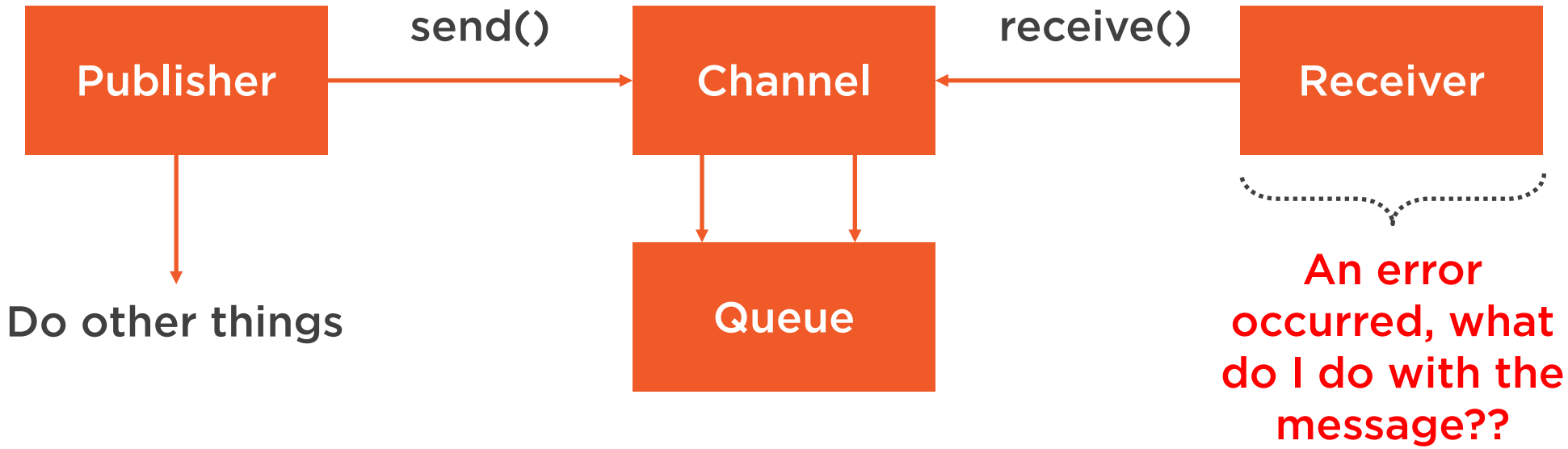
```
try {
        app.doSomething();
} catch (ApplicationException e) {
        // Handle the exception
}
```

Handling Exceptions in Synchronous Code

Execute code that can throw an exception in a try block

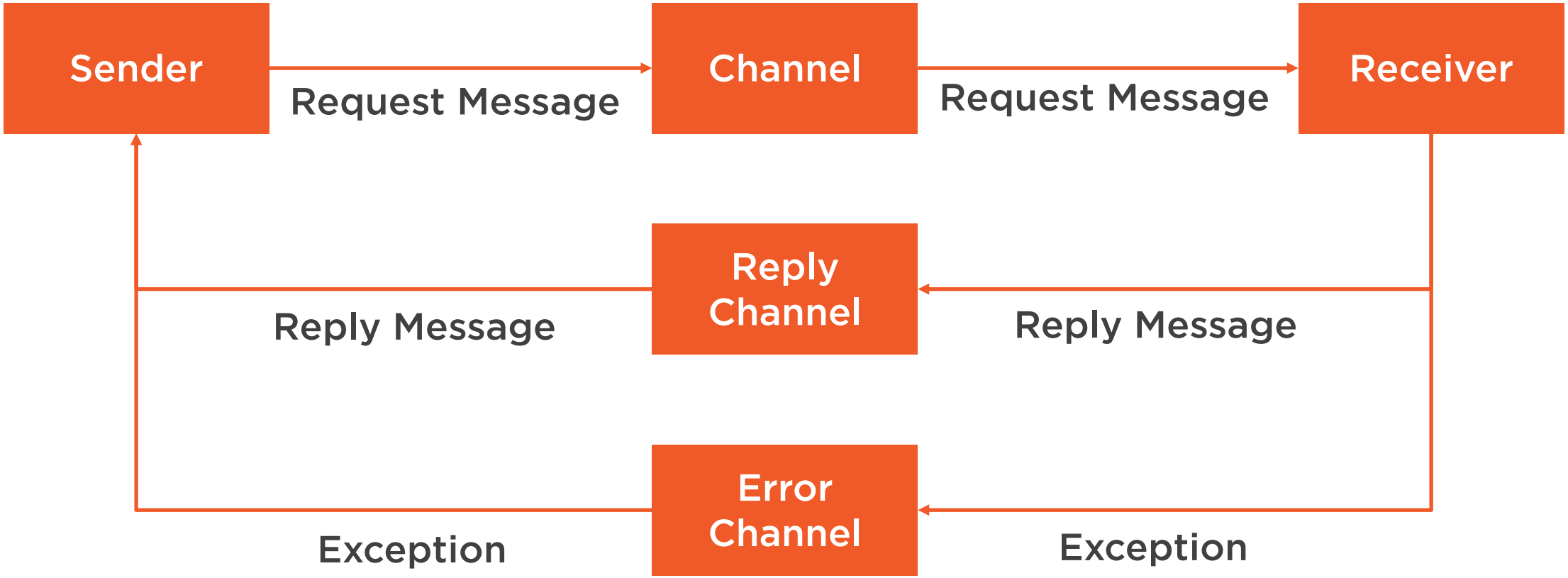
Catch exceptions in a catch block

Errors in Asynchronous Applications



Error Handling in a Request-Reply Message Pattern

Request-Reply Message Pattern



The sender sends a request message through a channel and the receiver sends the reply through a reply channel

Request-Reply Channels

Direct Channel

Sender and receiver share the same thread

Rendezvous Channel

Queuing Channel that uses a synchronous queue

```
public class RegistrationServiceImpl {
    @Autowired
    private HotelBookingGateway hotelBookingGateway;

    public Boolean checkAvailability(Integer numberOfGuests) {
        try {
          Message<Integer> message =
           MessageBuilder.withPayload(numberOfGuests).build();
          Message<Boolean> response =
              hotelBookingGateway.checkAvailability(message);
          return response.getPayload();
        } catch (Exception e) {
            logger.error("An error occurred", e);
            return false;
        }
    }
}

public class HotelBookingServiceImpl {
    @ServiceActivator(inputChannel = "hotelBookingChannel")
    public Message<Boolean> checkAvailability(
                          Message<Integer> numberOfGuests) {
        Integer guests = numberOfGuests.getPayload();
        if (guests <= 0) {
            throw new RuntimeException("Invalid party size: "
                                  + guests);
        }
        return MessageBuilder.withPayload(true).build();
    }
}
```

- Autowire Gateway
- Execute gateway method in a try block
- Handle errors in a catch block
- Throw a runtime exception

Demo

Define our components

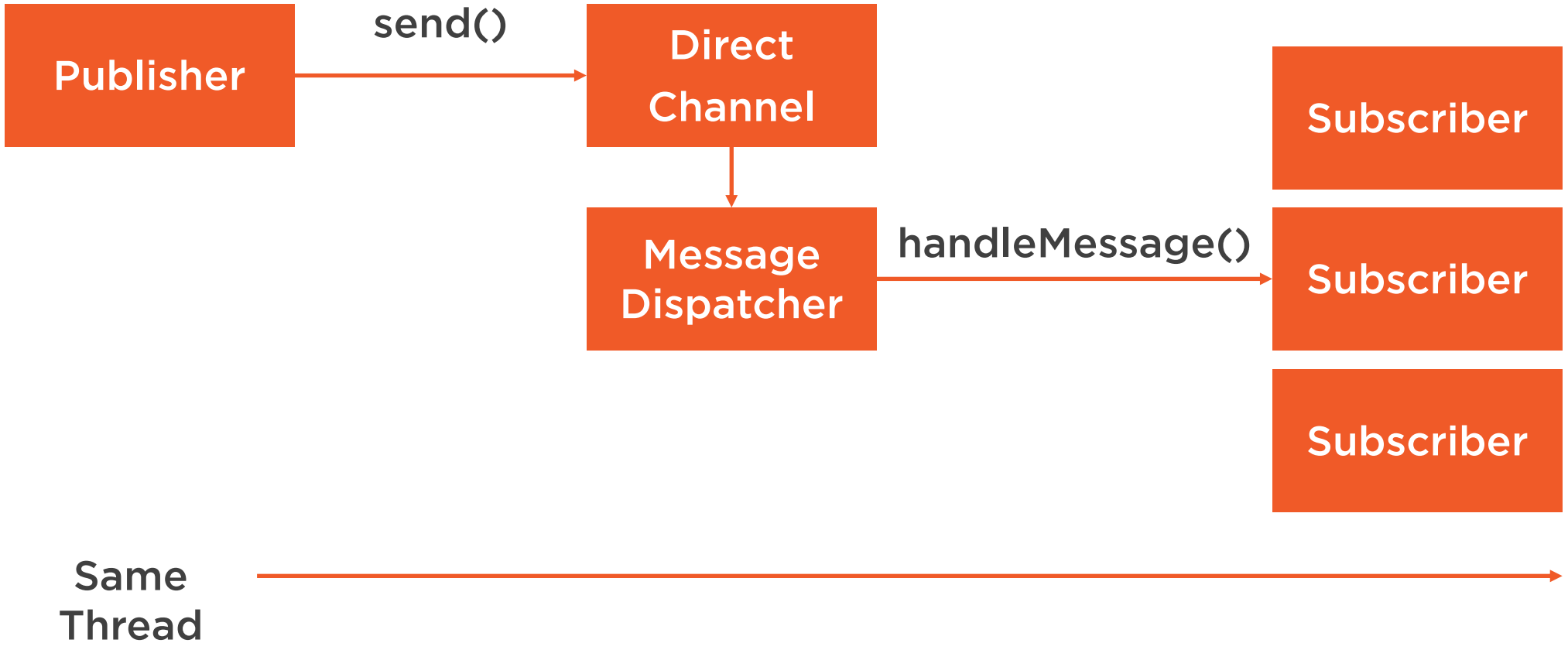
- Hotel Booking Channel (Direct Channel)
- Hotel Booking Gateway
- Registration Service
- Hotel Booking Service with service activator

Invoke the Registration Service

Throw an exception

Catch and log the exception

Direct Channel



Summary

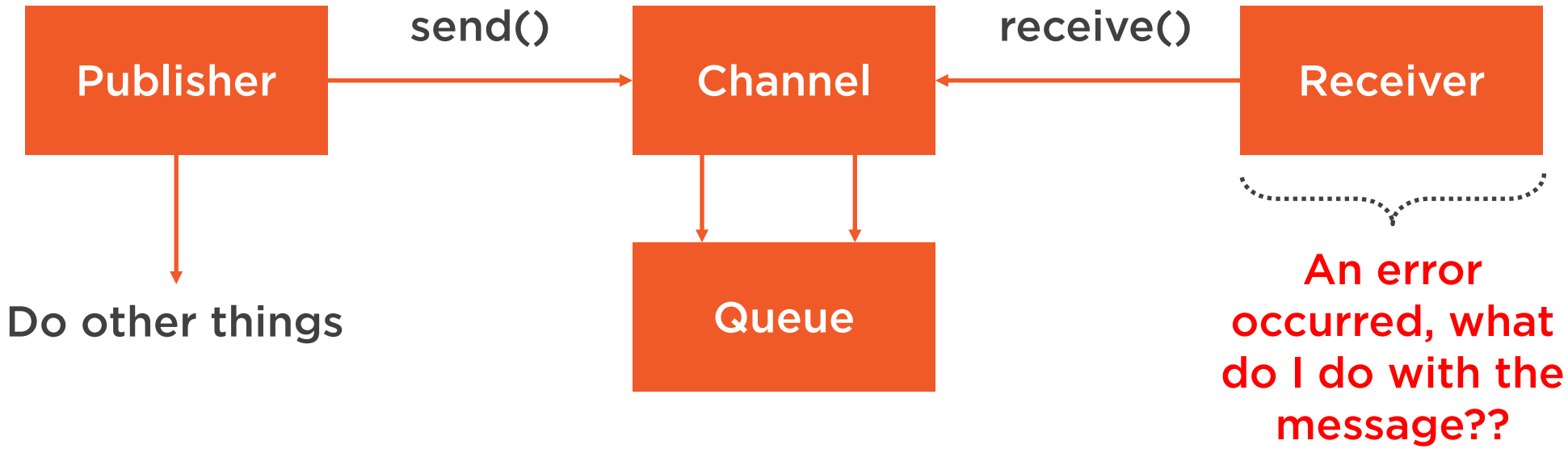
When using the request-reply message pattern, error handling can be accomplished using standard exception handling

Both Direct and Rendezvous channels work this way

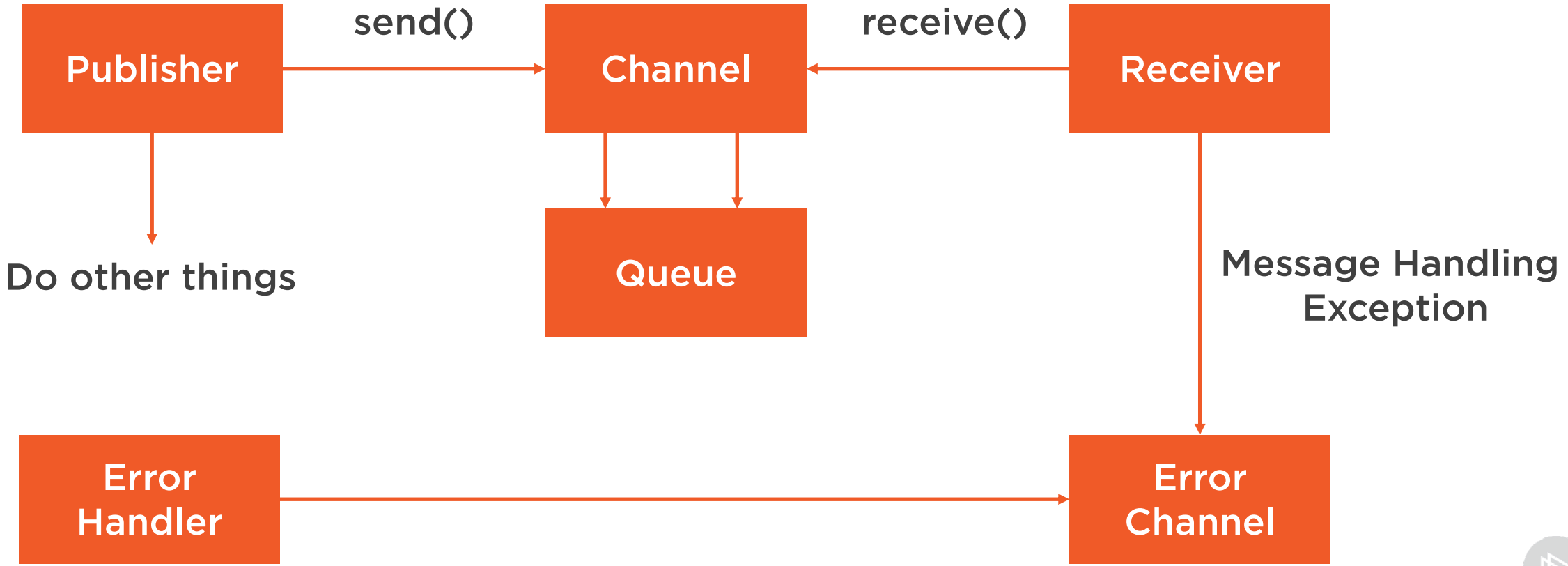
Next up: Asynchronous Message Error Handling

Error Handling in Asynchronous Message Channels

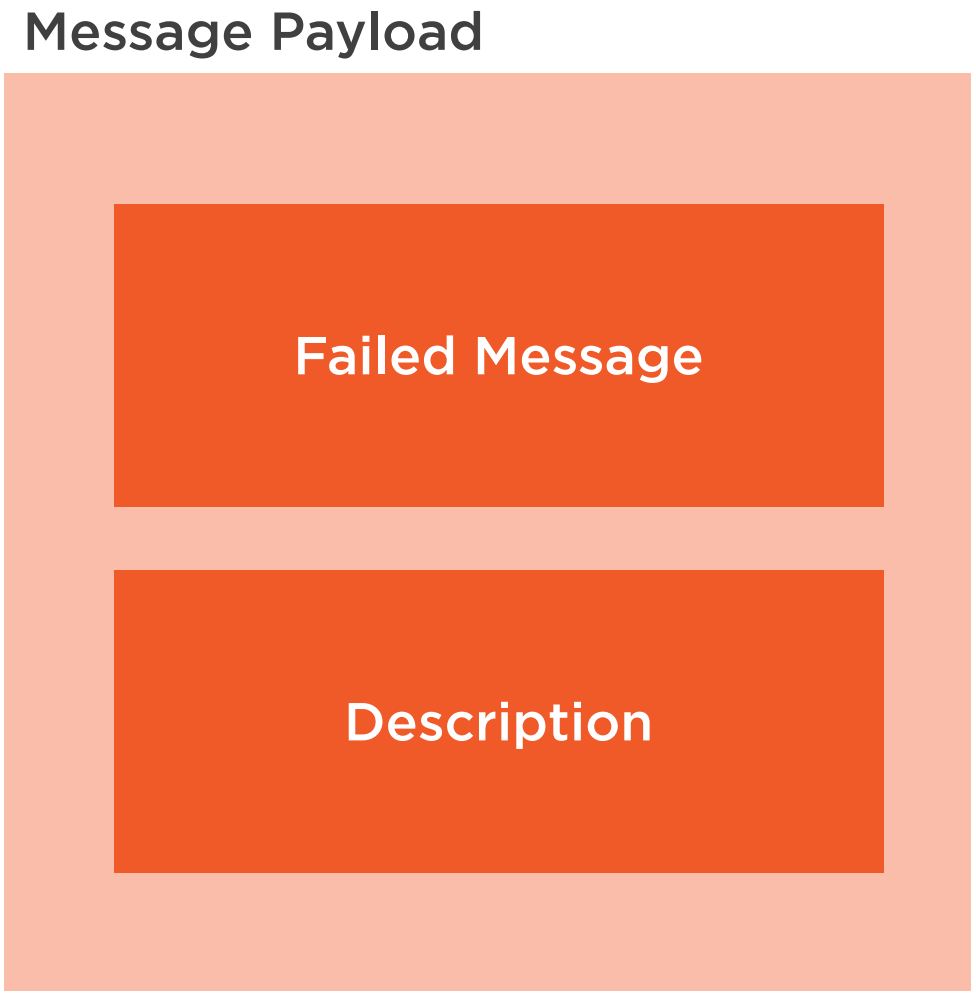
Errors in Asynchronous Applications



Errors in Asynchronous Applications



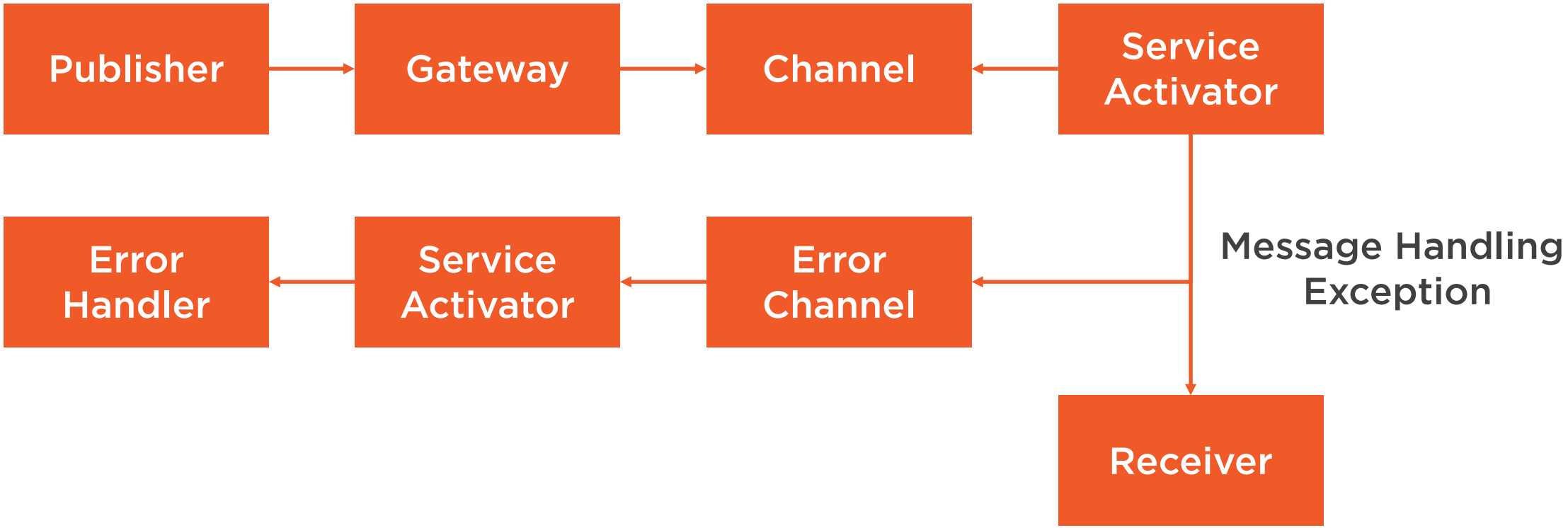
MessageHandlingException Message



The message that was being processed when an exception was thrown

A textual description of the exception

Error Handling Using a Gateway and Service Activator



Error Channels

Global Error Channel

Created by Spring Integration to catch all exceptions without an explicitly defined error channel

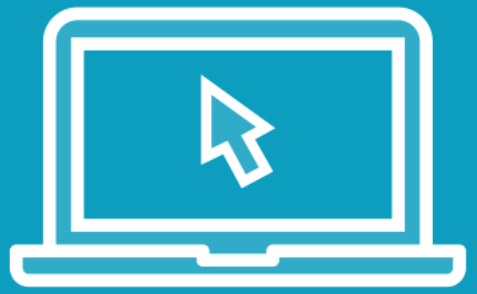
Custom Error Channel

An error channel that you create and specify in your messaging gateway

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

@ServiceActivator(inputChannel = "queueChannel",
                  poller = @Poller(fixedDelay = "100"))
public void sendSwag(Message<Swag> swag) {
   if (swag.getPayload().getType().equalsIgnoreCase("Hat")) {
     throw new RuntimeException("We're out of hats, sorry");
   }
}

@Service
public class ErrorChannelService {
  private static final Logger logger =
              LogManager.getLogger(ErrorChannelService.class);

  @ServiceActivator(inputChannel = "errorChannel")
  void handleException(Message<MessageHandlingException>
                                        message) {
    MessageHandlingException ex = message.getPayload();
    logger.error(”Exception : {}. {}",
                ex.getFailedMessage(),
                ex.getMessage());
    }
}
```

- Define a standard gateway
- Throw an exception if the registration service requests a hat
- Handle errors in a new Error Channel Service

Demo



Define our components

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

Invoke the Registration Service

Throw an exception

Handle the failed message in the Error Channel Service

Summary

When an asynchronous message receiver cannot handle a message, it throws an exception that gets published to an error channel

There are two types of error channels: the global error channel and custom error channels

Next up: Custom Error Channels

Custom Error Channels

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

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

@ServiceActivator(inputChannel = "queueChannel",
                  poller = @Poller(fixedDelay = "100"))
public void sendSwag(Message<Swag> swag) {
  if (swag.getPayload().getType().equalsIgnoreCase("Hat")) {
    throw new RuntimeException("We're out of hats, sorry");
  }
}

public class ErrorChannelService {
  @ServiceActivator(inputChannel = “customErrorChannel")
  void handleException(Message<MessageHandlingException> m) {
    MessageHandlingException ex = m.getPayload();
    logger.error(”Exception: {}. {}",
                 ex.getFailedMessage(),
                 ex.getMessage());
    }
}
```

- Define a custom error channel
- Configure the messaging gateway to publish errors to our custom error channel
- Throw an exception if the registration service requests a hat
- Handle errors from our custom error channel

Demo

Define our components

- Queue Channel
- Custom Error Channel
- Queue Channel Gateway
- Registration Service
- Swag Service with service activator
- Error Channel Service with service activator

Invoke the Registration Service

Throw an exception

Handle the failed message in the Error Channel Service

Summary

A custom error channel is defined in your configuration class as a PublishSubscribeChannel

The gateway is configured to publish errors to the custom error channel through its “errorChannel” property

The error handler is subscribed to the custom error channel

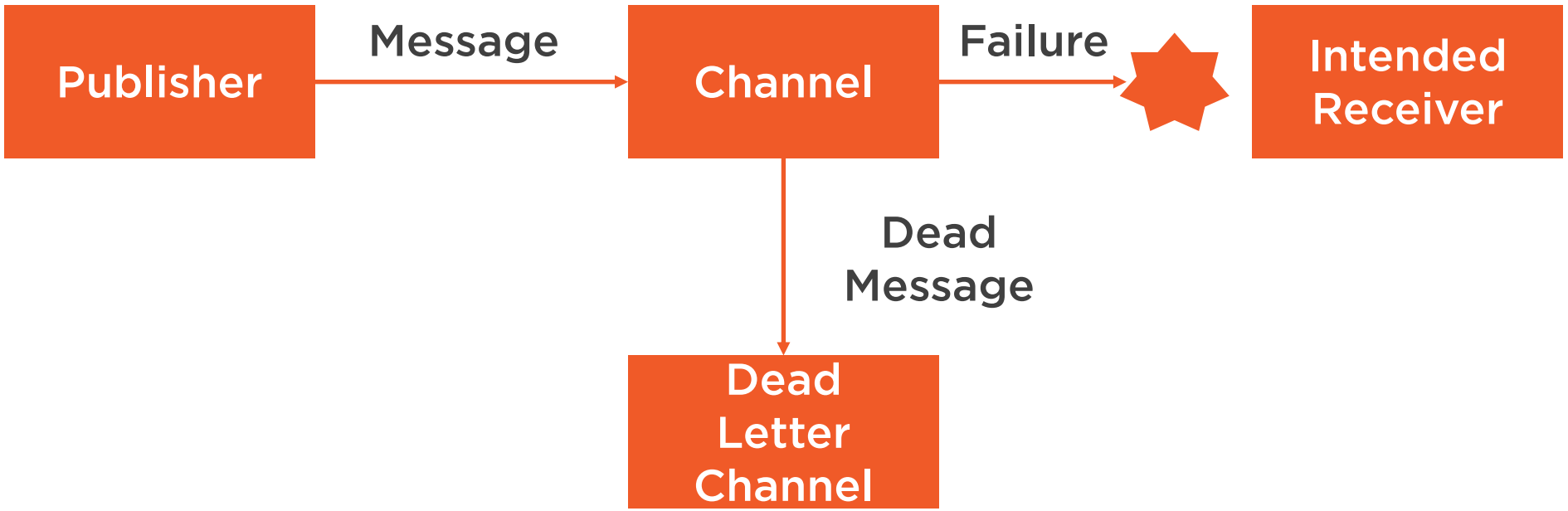
Next up: Overview of Dead Letter Channels

Overview of Dead Letter Channels

Dead Letter Channel

When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a Dead Letter Channel

Dead Letter Channel



Example: Amazon SQS

When a consumer retrieves a message from a queue, the message remains hidden on the queue. After processing the message, the consumer is responsible for deleting the message. If it does not delete the message in the allotted time, it becomes visible again. Once a message has been retrieved a certain number of times, SQS moves it to a dead letter queue.

Conclusion

Handling Errors

Synchronous

Handling errors when the message publisher is available to receive the error

Asynchronous

Handling errors when the message publisher is no longer available to receive the error

Spring Integration Error Channels

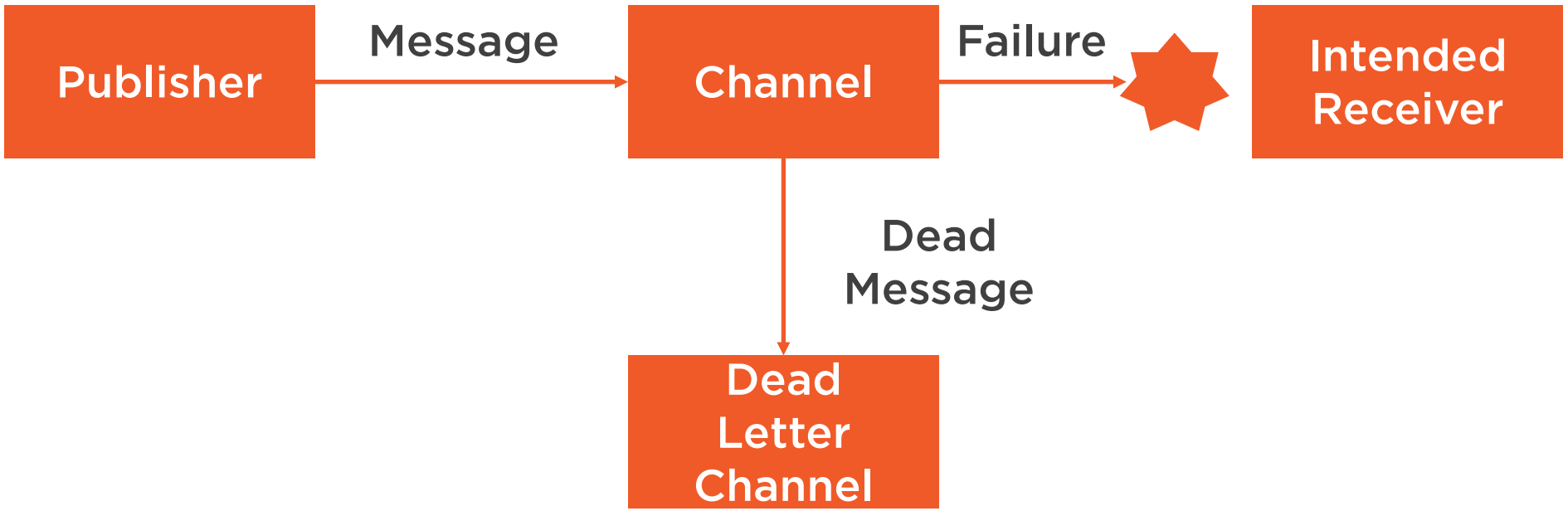
Global Error Channel

Created by Spring Integration to catch all exceptions without an explicitly defined error channel

Custom Error Channel

An error channel that you create and specify in your messaging gateway

Dead Letter Channel



Summary

You should now understand how to handle errors in your Spring Integration applications

You should feel comfortable using channels in a robust and resilient manner to build better Spring Integration applications