# Spring Integration: Message System Design Patterns

## DESIGNING MESSAGES FOR A MESSAGING SOLUTION

**Steven Haines**
PRINCIPAL SOFTWARE ARCHITECT
@geekcap www.geekcap.com

## Overview

Enterprise Integration Patterns

Command Message

Document Message

Request-Reply Message

Event Message

## Enterprise Integration Patterns

65 patterns that provide technology-independent design guidance for developers and architects to describe and develop robust integration solutions.

https://www.enterpriseintegrationpatterns.com

## Message Construction Patterns

Command Message

Document Message

Request-Reply Message

Event Message

## Module Overview

What it is

How it works

Comparison

Demonstration

## Command Messages

## Overview

Definition of a Command Message

How the Command Message works

When to use a Command Message

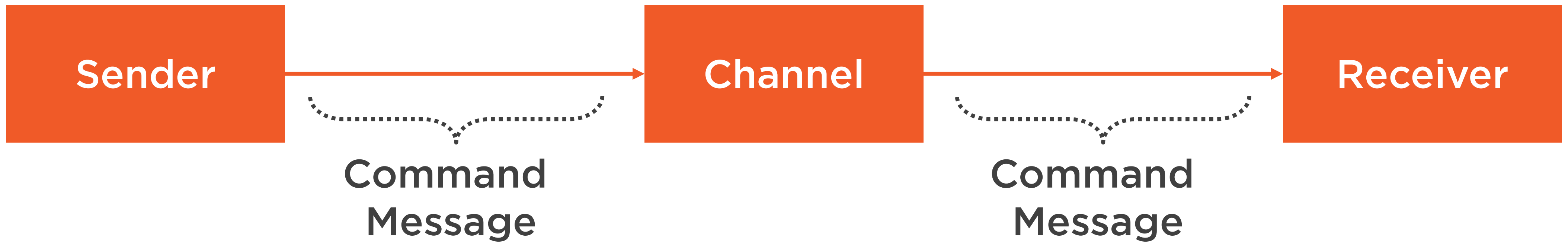
How it is implemented in Spring Integration

Demo

## Command Message

A Command Message is a message used when an application needs to invoke functionality provided by another application.

# How the Command Message Works

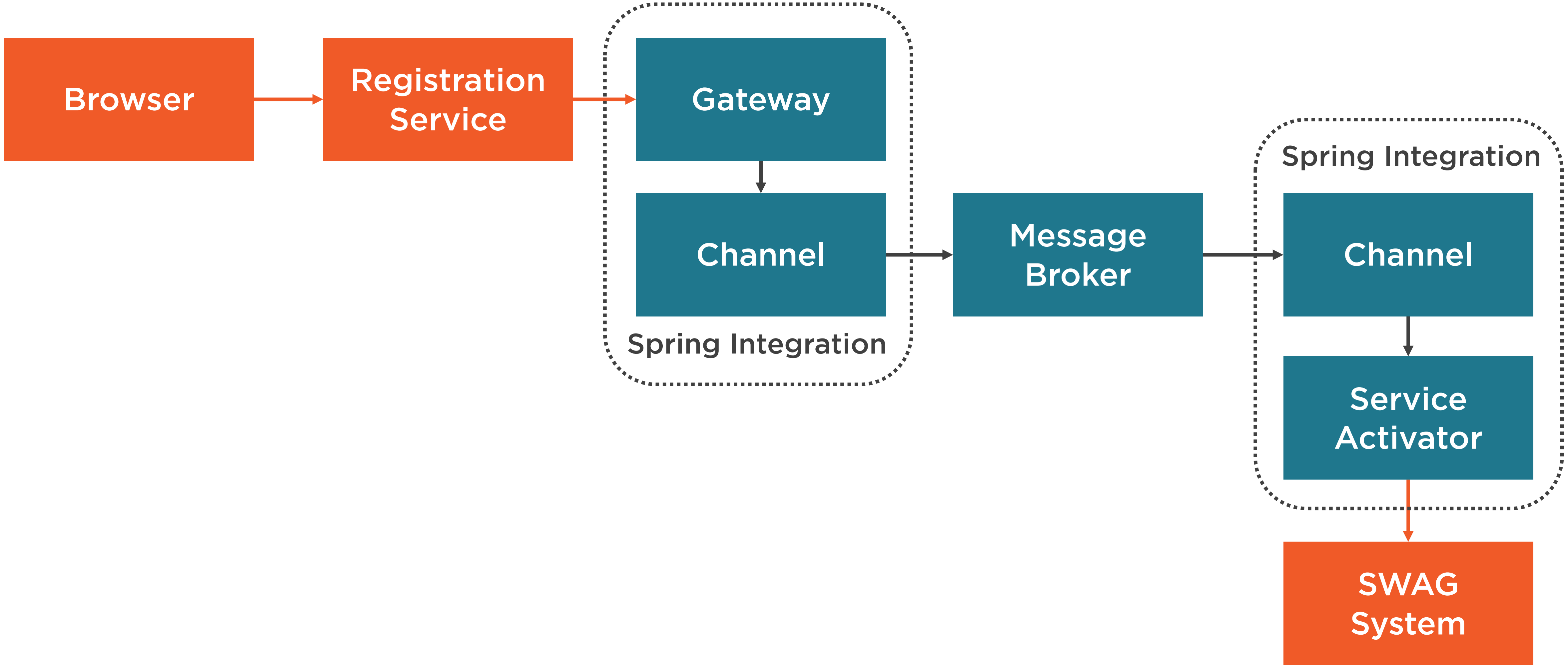


A Command Message is a regular message that contains a command

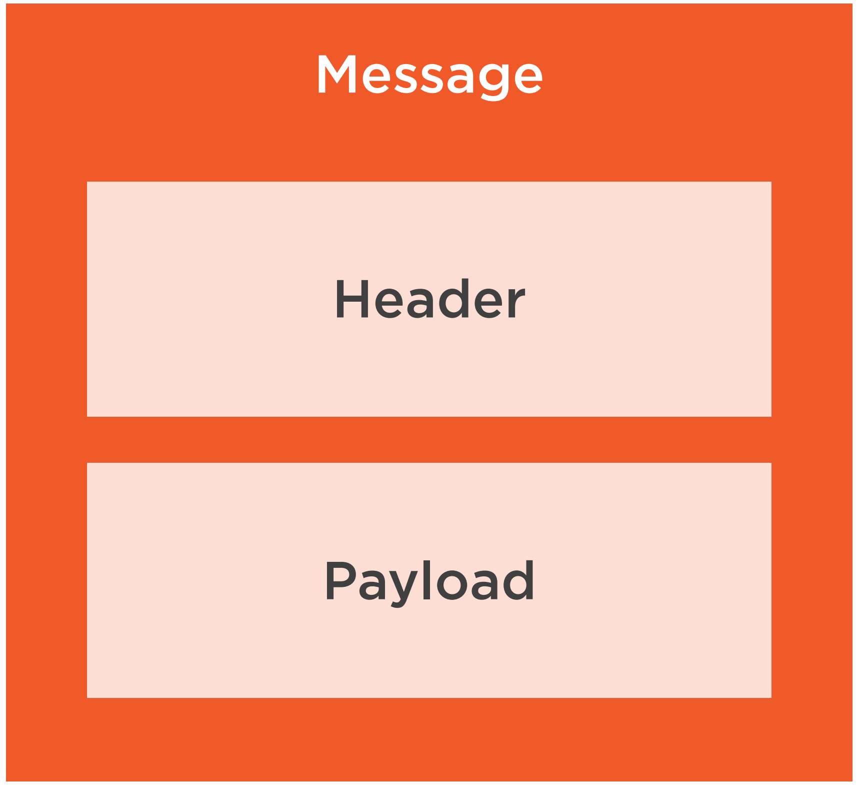
# Why Use the Command Pattern?

| Remote Procedure Call | Command Message |
|---|---|
| Tightly couples systems | Loosely couples systems |
| Requires the receiver to be available, otherwise the call will fail | Receiver can be offline and will process the message when it is restored |
| List of receivers must be known at build time | Receivers can change over time without requiring a change to the sender |

## Example: Publishing a Command Message



## Messages



The Header contains additional information about the message, such as a correlation ID, sequence ID, expiration time

The Payload contains the body of the message

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

@MessagingGateway(name="swagGateway",
        defaultRequestChannel="swagChannel")
public interface SwagGateway {
    @Gateway
    void sendSwag(Message<Swag> swag);
}

@Service
public class RegistrationServiceImpl {
    @Autowired
    private SwagGateway swagGateway;
    public void commit(String userId) {
        Message<Swag> message = MessageBuilder
           .withPayload(new Swag("T-Shirt")).build();
        swagGateway.sendSwag(message);
    }
}

@MessageEndpoint
@Service
public class SwagServiceImpl implements SwagService {
    @ServiceActivator(inputChannel= "swagChannel")
    public void sendSwag(Message<Swag> swag) {
        logger.info("SwagService::Sending Swag: {}", message.getPayload());
    }
}
```

◂ Define a channel

◂ Define a Gateway

◂ Publish to the Gateway

◂ Handle the message

## Demo

**Define our components**

- Swag Channel
- Swag Gateway
- Registration Service
- Swag Service with a service activator

**Invoke the Registration Service to complete a registration**

**Publish a message using the Swag Gateway**

**Validate that the Swag Service is executed with the Command Message**

## Summary

A Command Message is a normal message that is used to invoke the functionality of another component

It enforced loose coupling and helps future proof the integration of systems over normal RPC calls

Next up: Document Messages

## Document Messages

## Overview

Definition of a Document Message

How the Document Message works

When to use Document Messages

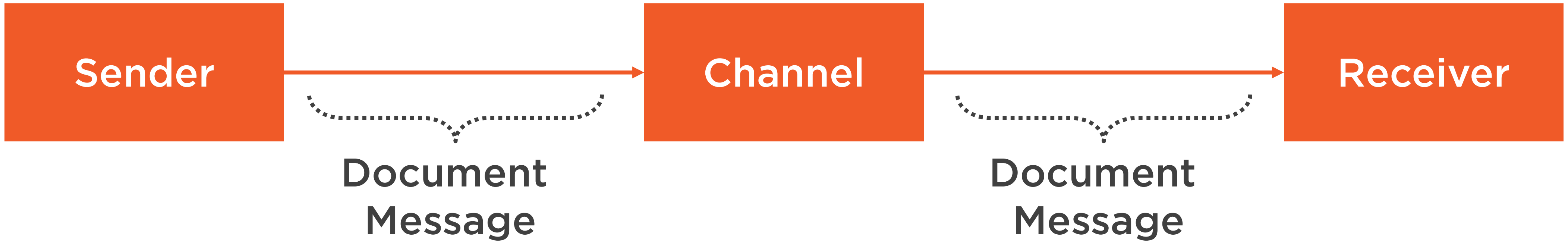
How it is implemented in Spring Integration

Demo

## Document Message

A Document Message is used when an application would like to transfer data to another application.

## How the Document Message Works

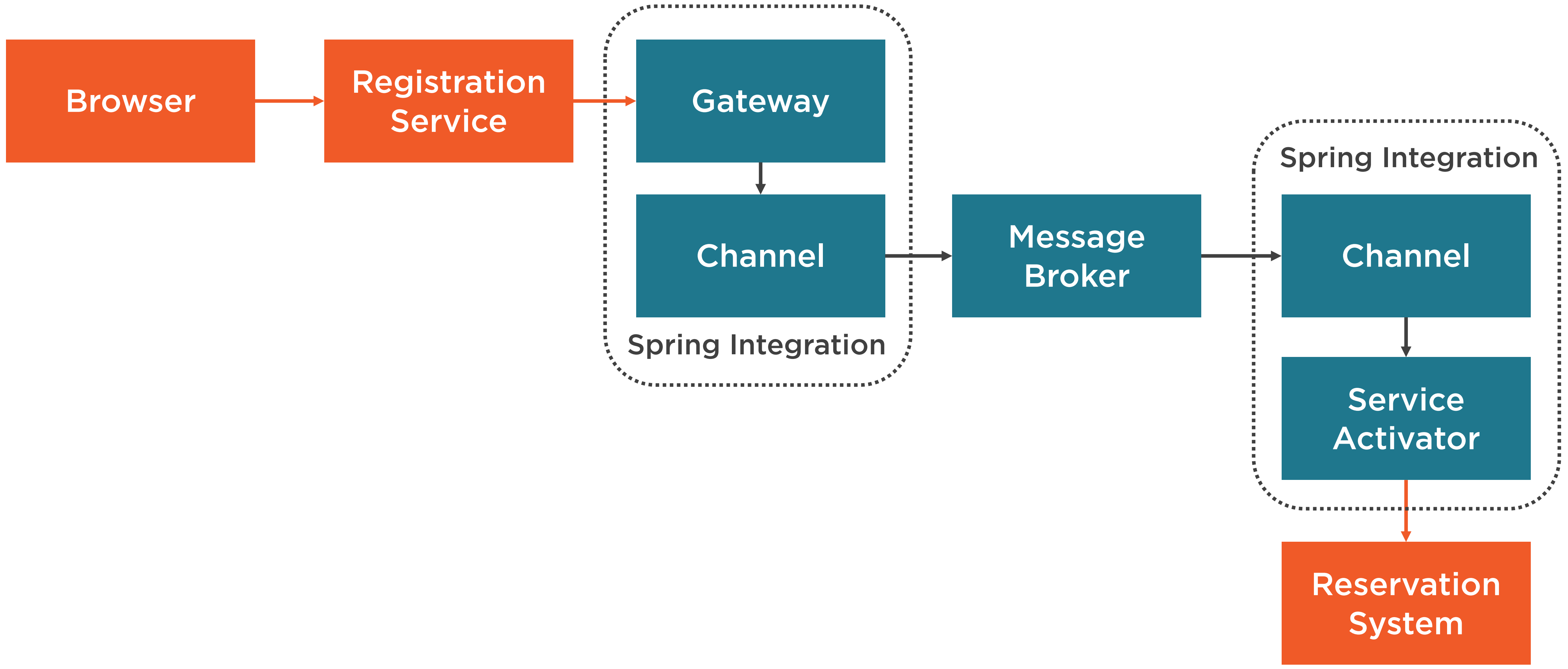


A Document Message can contain a single piece of data or a data structure which may decompose into smaller pieces of data

# Why Use the Document Message Pattern?

| File Transfer or Shared Database | Document Message |
|---|---|
| Tightly couples systems | Loosely couples systems |
| Requires the receiver (or database) to be available, otherwise the call will fail | Receiver can be offline and will process the message when it is restored |
| List of receivers must be known at build time | Receivers can change over time without requiring a change to the sender |

# Example: Publishing a Document Message



## Message Headers

### Sequence Number

The sequence number of this message in a group of messages

### Sequence Size

The number of messages within a group of correlated messages

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

◀ Define a channel

```
@MessagingGateway(name=”reservationRecordGateway",
        defaultRequestChannel=”reservationRecordChannel")
public interface ReservationRecordGateway {
    @Gateway
    void addRecord(Message<ReservationRecord> record);
}
```

◀ Define a Gateway

```
@Service
public class RegistrationServiceImpl {
    @Autowired
    private ReservationRecordGateway reservationRecordGateway;
    public void updateReservationRecord(
                    ReservationRecord record) {
        Message<ReservationRecord> message = MessageBuilder.withPayload(record)
                .setHeader(IntegrationMessageHeaderAccessor.SEQUENCE_NUMBER, 1)
                .setHeader(IntegrationMessageHeaderAccessor.SEQUENCE_SIZE, 5)
                .build();
        reservationRecordGateway.addRecord(message);
    }
}
```

◀ Publish to the Gateway

```
@MessageEndpoint
@Service
public class ReservationServiceImpl implements ReservationService {
   @ServiceActivator(inputChannel=”reservationRecordChannel")
   public void addRecord(Message<ReservationRecord> record) {
        IntegrationMessageHeaderAccessor accessor = new
                                                       IntegrationMessageHeaderAccessor(message);
        logger.info("Sequence: {} / {}", accessor.getSequenceNumber(),
                                                 accessor.getSequenceSize());
        logger.info("Add reservation record: {}", message.getPayload());
   }
}
```

◀ Handle the message

## Demo

**Define our components**

- Reservation Record Channel
- Reservation Record Gateway
- Registration Service
- Reservation Service with a service activator

**Invoke the Registration Service to add a reservation record**

**Publish a message using the Reservation Record Gateway**

**Validate that the Reservation Service is executed with the Document Message**

## Summary

A Document Message is used to transfer data to another component

It enforced loose coupling and helps future proof the integration of systems over normal file transfer or shared database options

Next up: Request-Reply Messages

## Request-Reply Messages

## Overview

Definition of a Request-Reply Message

How the Request-Reply Message works

When to use Request-Reply Messages

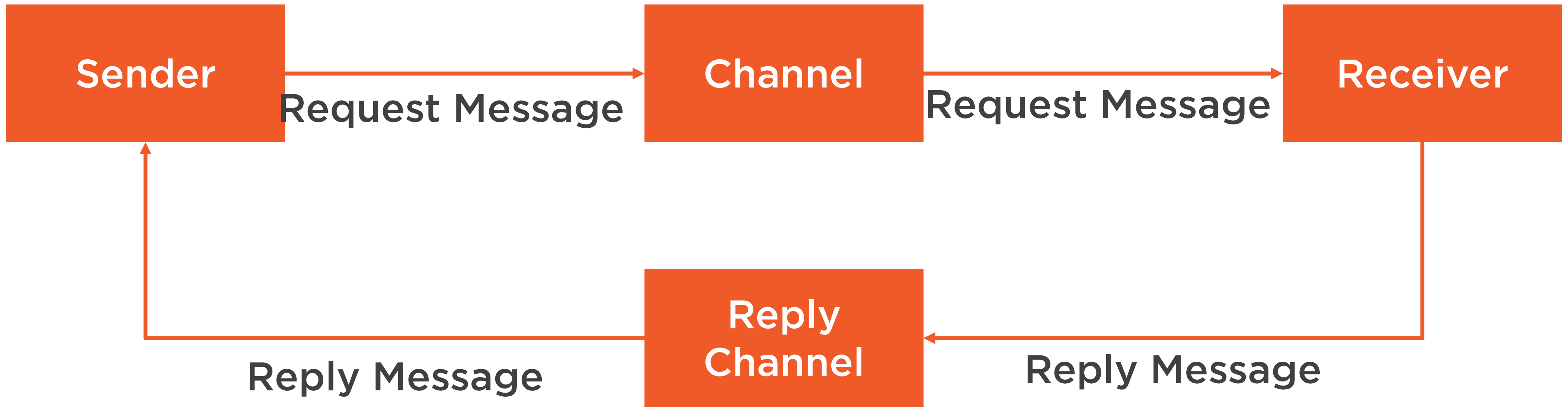
How it is implemented in Spring Integration

Demo

## Request-Reply Message

A Request-Reply Message is used to facilitate a two-way conversation via messaging.

## How the Request-Reply Message Works

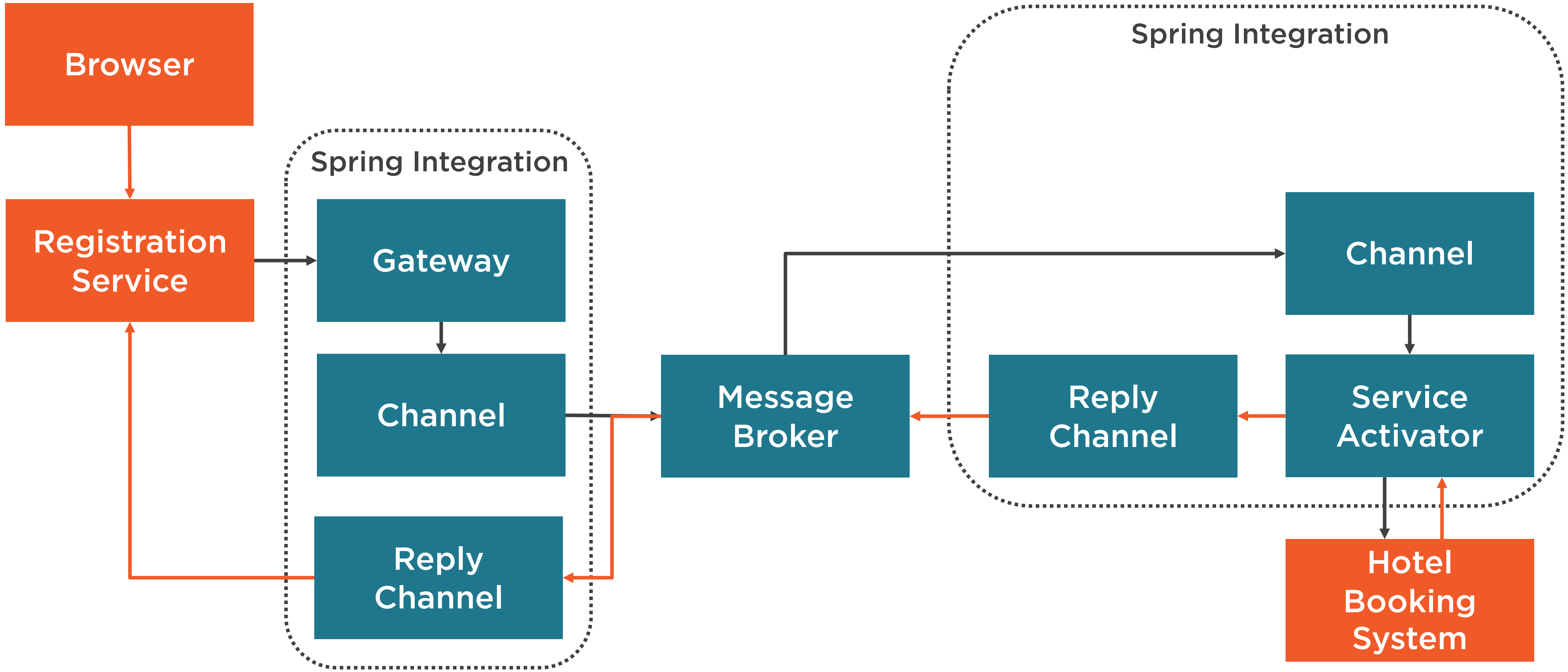


The sender sends a request message through a channel and the receiver sends the reply through a reply channel

## Why Use the Request-Reply Message Pattern?

| Web Service or RPC call | Request-Reply Message |
|---|---|
| Tightly couples systems | Loosely couples systems |
| Receiver must be known at build time | Receiver can change over time without requiring a change to the sender |

# Example: Publishing a Request-Reply Message



# Automatically Created Message Headers

## replyChannel

When the gateway method returns a result, Spring Integration automatically creates a temporary reply channel and the service activator will publish the result to the reply channel

## errorChannel

If the service activator method throws an exception, then the service activator publishes the exception to the error channel

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

◀ Define a channel

```
@MessagingGateway(name=”hotelBookingGateway",
        defaultRequestChannel=”hotelBookingChannel")
public interface HotelBookingGateway {
    @Gateway
    Message<Boolean> checkAvailability(Message<Integer> numberOfGuests);
}
```

◀ Define a Gateway

```
@Service
public class RegistrationServiceImpl {
    @Autowired
    private HotelBookingGateway hotelBookingGateway;
    public Boolean checkAvailability(Integer numberOfGuests) {
        Message<Integer> message = MessageBuilder
                             .withPayload(numberOfGuests).build();
        Message<Boolean> response =
                   hotelBookingGateway.checkAvailability(message);
        return response.getPayload();
    }
}
```

◀ Publish to the Gateway and receive the response

```
@MessageEndpoint
@Service
public class HotelBookingServiceImpl implements HotelBookingService {
   @ServiceActivator(inputChannel = "hotelBookingChannel")
   public Message<Boolean> checkAvailability(Message<Integer> numGuests) {
       Integer guests = numberOfGuests.getPayload();
       return MessageBuilder.withPayload(true).build();
   }
}
```

◀ Handle the message and return a response

## Demo

**Define our components**

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

**Invoke the Registration Service to check availability**

**Publish a message using the Hotel Booking Gateway**

**Validate that the Hotel Booking Service is executed and return a response**

## Summary

A Request-Reply Message is used to facilitate two-way communication using messaging

It enforces loose coupling and helps future proof the integration of systems over RPC or Web Service calls

Next up: Event Messages

## Event Messages

## Overview

Definition of an Event Message

How the Event Message Works

When to use Event Messages

Heavyweight vs. Lightweight Events

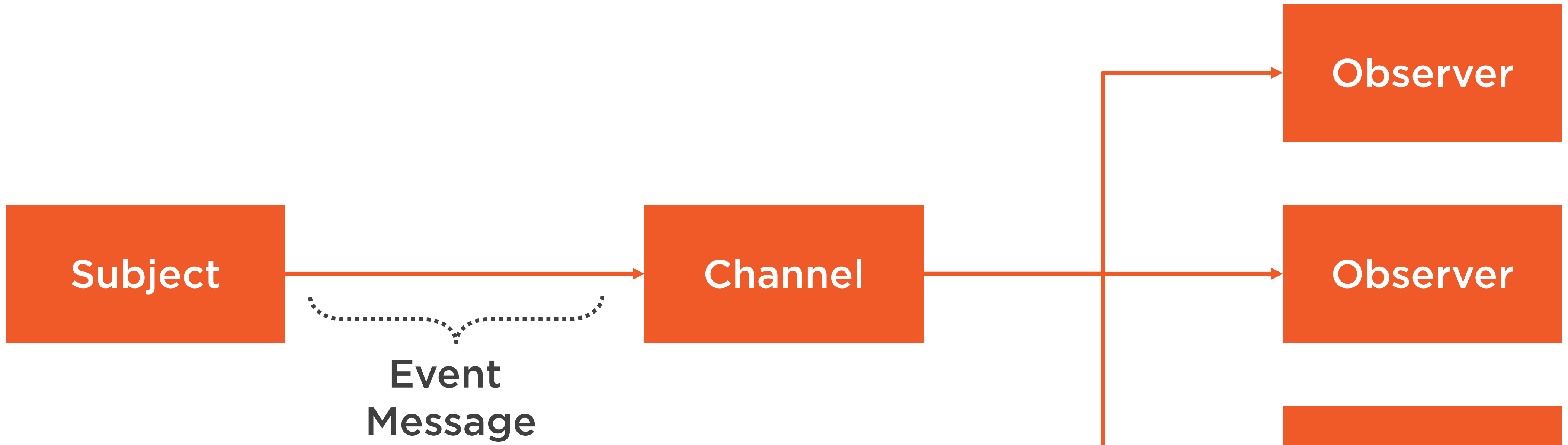
How it is implemented in Spring Integration

Demo

## Event Messages

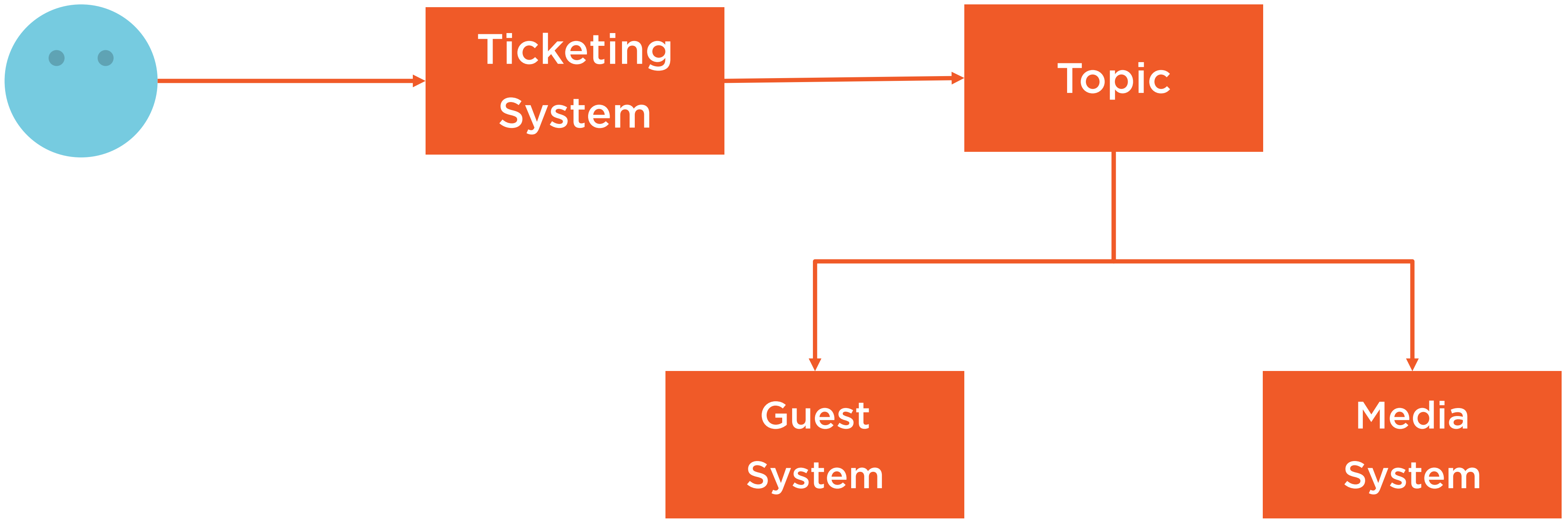
An application publishes events when its internal state has changed. Other applications can integrate with that application by listening to its events.

## Event-driven Architecture



A Subject publishes an event to a channel.
Observers receive the event and process it

# Real-world Example: Ticketing System



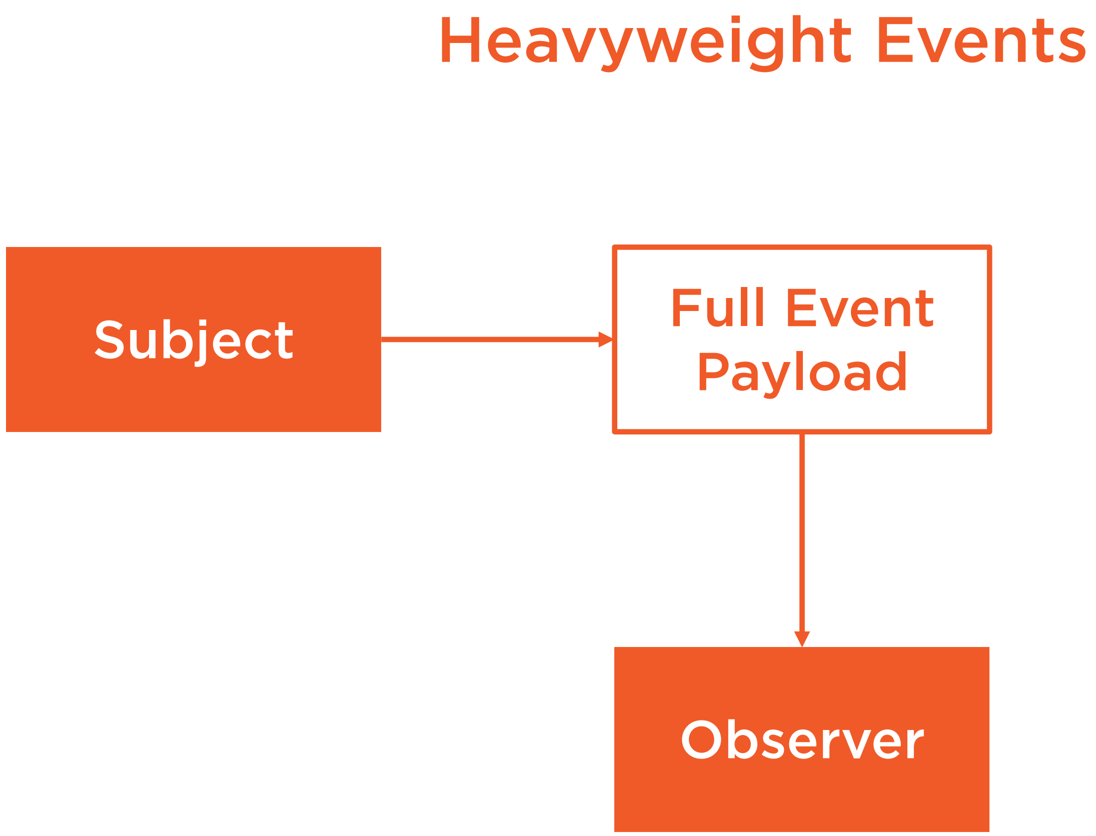
## Event-driven Architecture

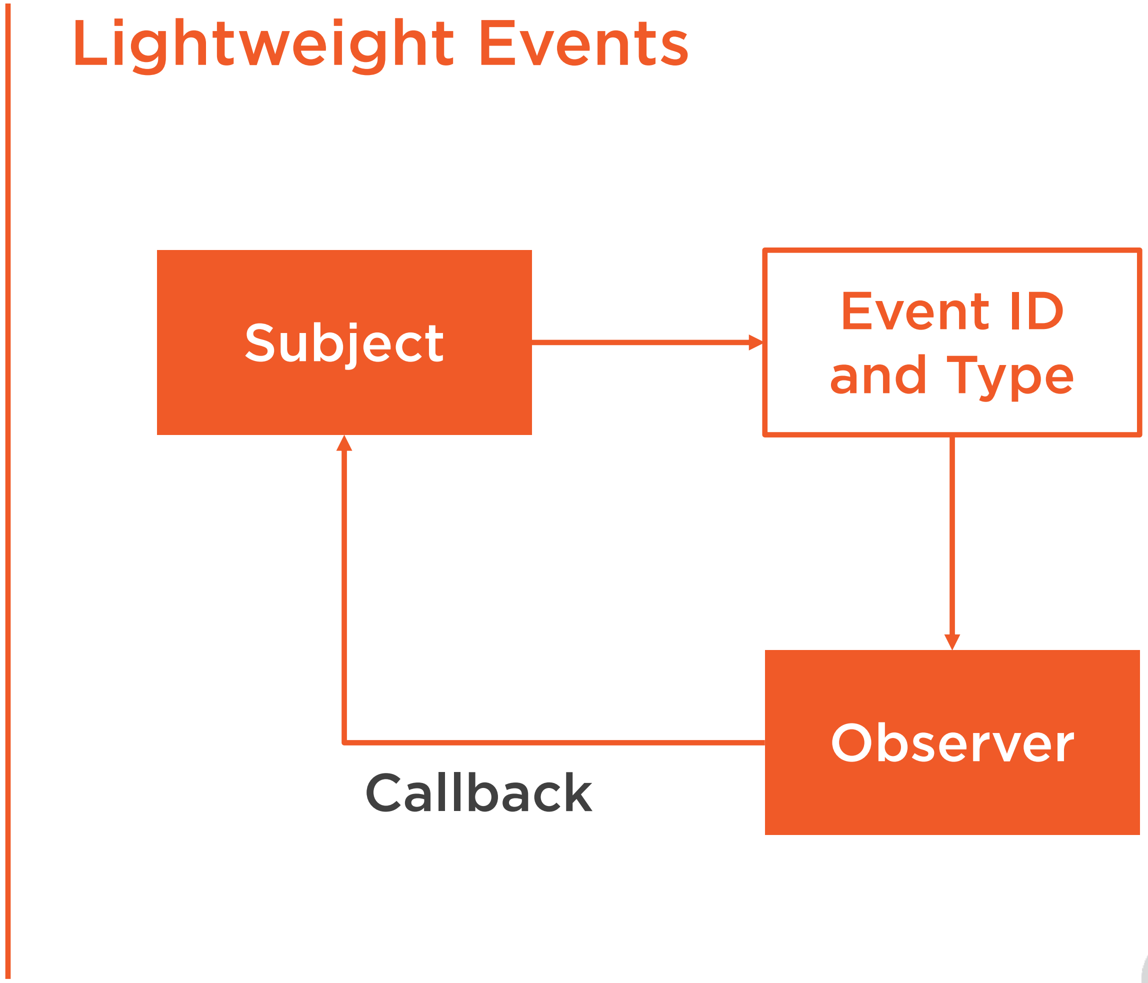
**Highly Scalable**
A system can integrate with other systems asynchronously

**Loosely Coupled**
A system does not need to know about the other systems with which it is integrating

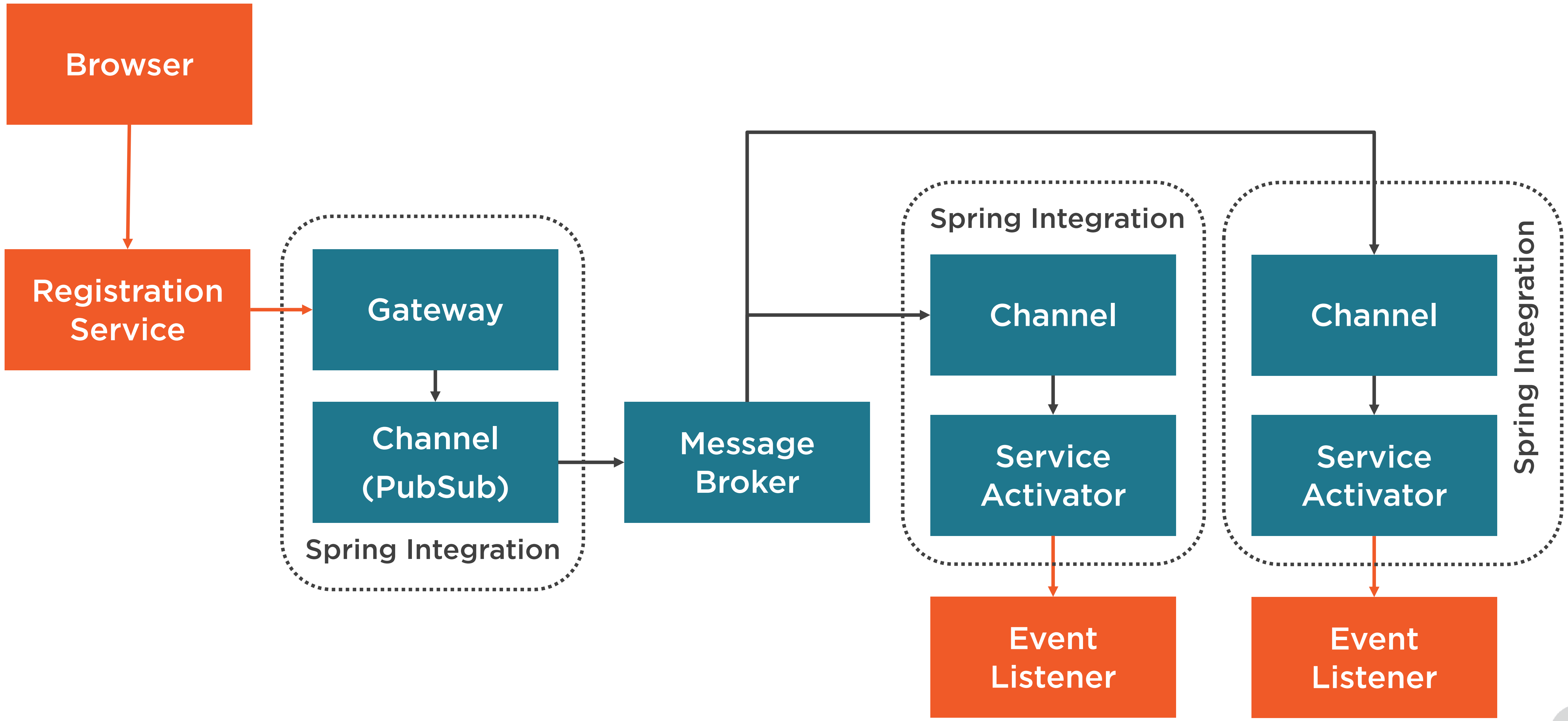
**Eventually Consistent**
Systems will eventually contain the same data

# Heavyweight vs. Lightweight Events





# Example: Publishing an Event 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**

## Demo

**Define our components**

- Event Channel
- Event Gateway
- Registration Service
- Event Listeners with service activators

**Invoke the Registration Service to notify observers**

**Publish a message using the Event Gateway**

**Validate that the Event Listeners are executed with the Event Message**

## Summary

An Event Message is used to allow one application to notify other applications of changes to its internal state

It enforces loose coupling and allows other applications to integrate with it

## Conclusion

## Enterprise Integration Patterns

65 patterns that provide technology-independent design guidance for developers and architects to describe and develop robust integration solutions.

https://www.enterpriseintegrationpatterns.com

# Message Construction Patterns

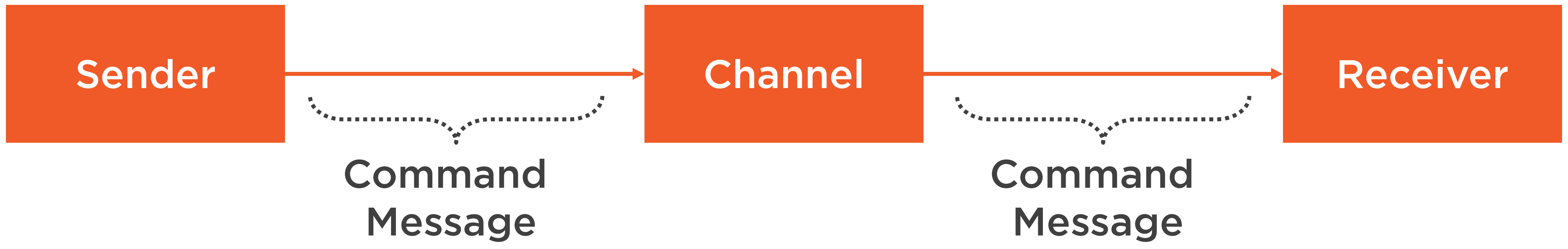
Command Message

Document Message

Request-Reply Message

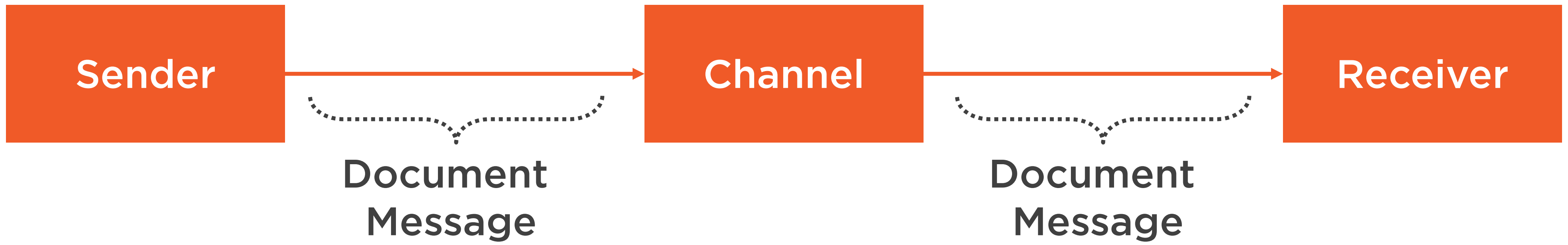
Event Message

# How the Command Message Works



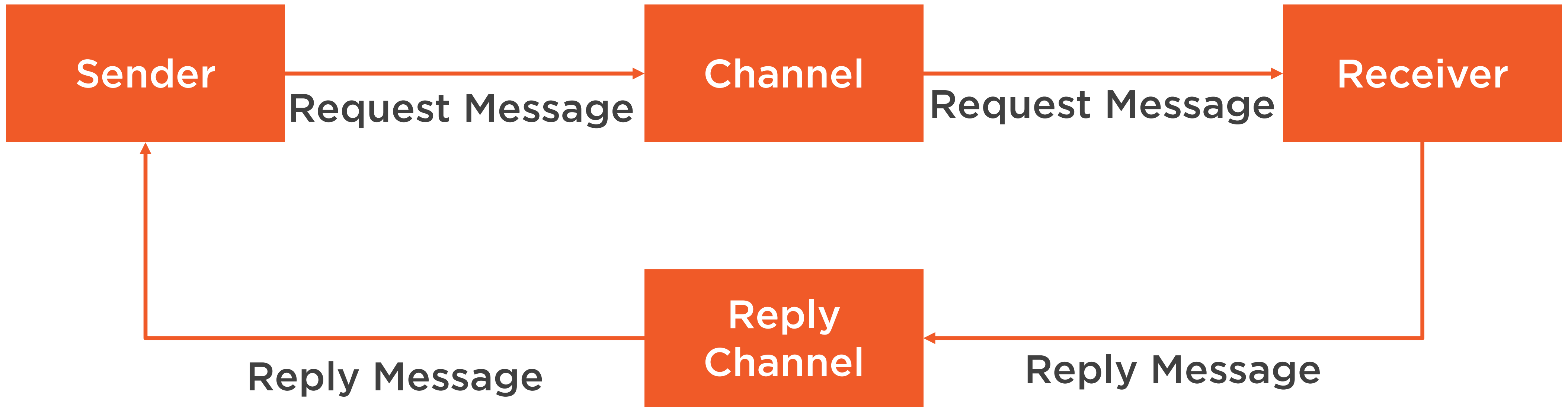
A Command Message is a regular message that contains a command

## How the Document Message Works



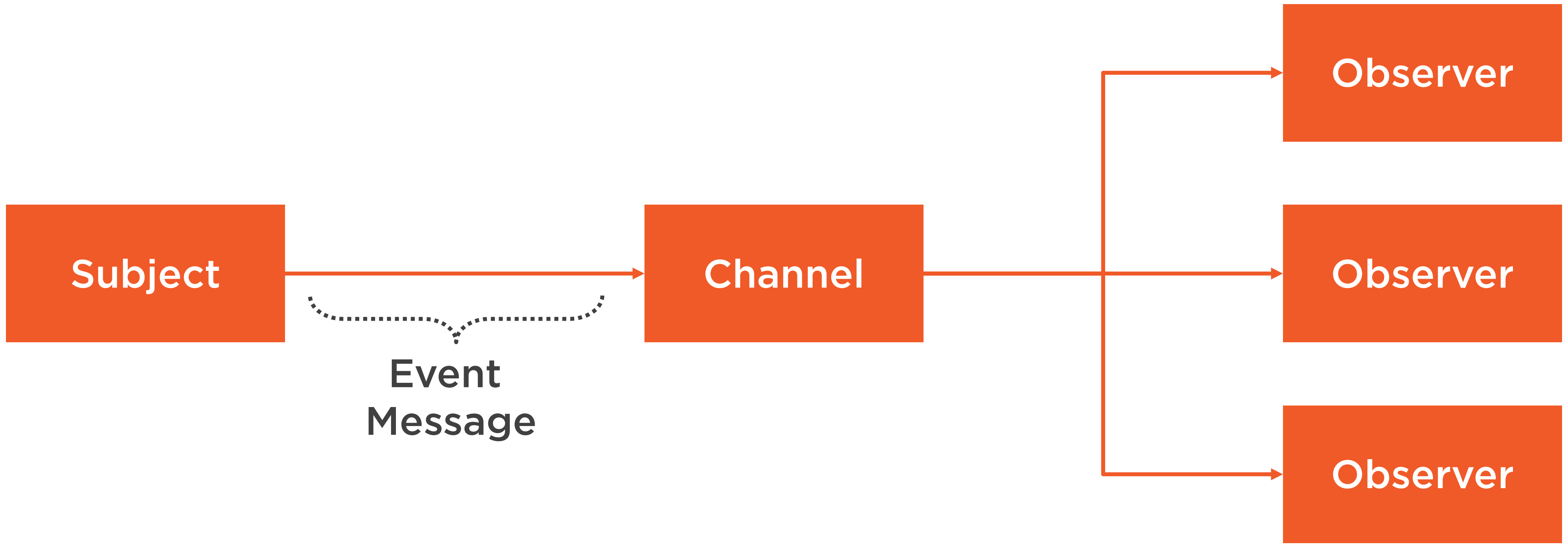
A Document Message can contain a single piece of data or a data structure which may decompose into smaller pieces of data

## How the Request-Reply Message Works



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

# Event-driven Architecture



A Subject publishes an event to a channel.
Observers receive the event and process it

## Summary

You should now understand the Common Message construction patterns

You should feel comfortable with Channels, Gateways, and service activators

You should be able to implement the Command Message, Document Message, Request-Reply Message, and Event Message patterns using Spring Integration