Connecting Microservices Through Messaging

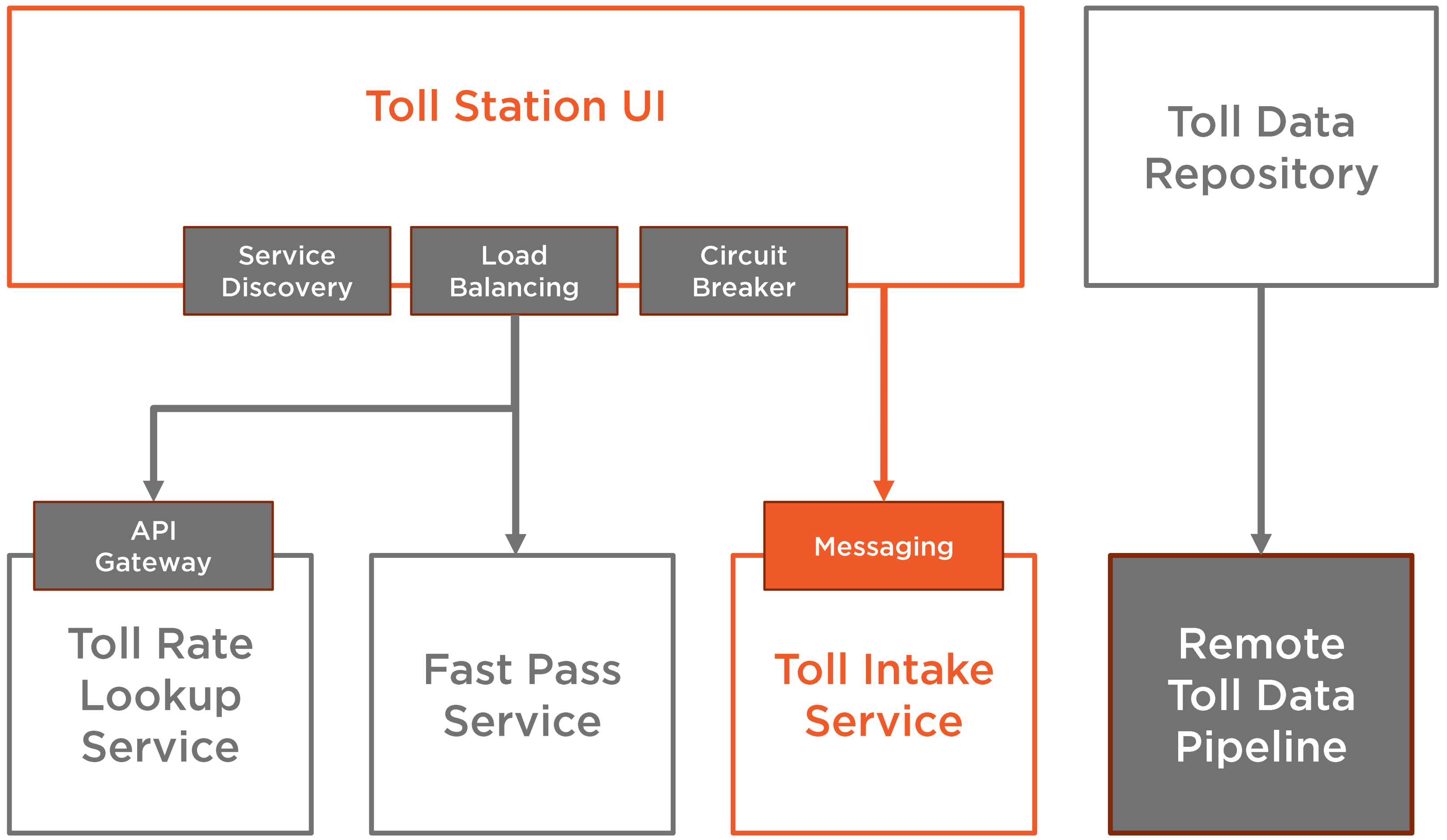
Richard Seroter
SENIOR DIRECTOR OF PRODUCT, PIVOTAL

@rseroter

Overview

- Role of messaging in microservices
- Problems with the status quo
- Describing Spring Cloud Stream
- Creating Stream applications
- The role of processors
- Using Consumer Groups to scale
- Doing stateful processing with partitions
- Working with content types
- Health and monitoring
- Summary

Capabilities That We Will Add in This Module



The Role of Messaging in Microservices

Encourage loose coupling

Improve scalability and reliability

Introduce new intake and processing patterns

Interact with legacy systems

Problems with the Status Quo

Messaging is often the realm of experts

Tight coupling creates fragility

Fixed flows and rigid endpoints

Legacy tools don’t make event-driven architecture easy

Spring Cloud Stream

Framework for building message-driven microservices apps.

Relationship with Spring Integration

MessageChannel and Message<T>

Channel Adapters

ServiceActivator

Spring Cloud Stream Core Concepts

Apps communicate through channels

Middleware abstracted via binders

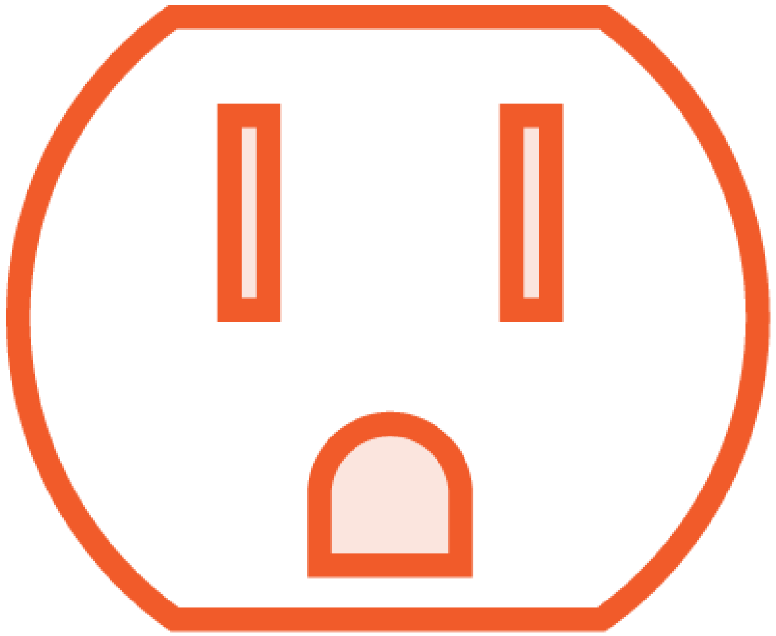
@StreamListener to pull events

Pub/sub pattern

Consumer groups for competing consumer

Partitioning for stateful processing

Explaining Binders



Connects you to physical endpoints in the external middleware

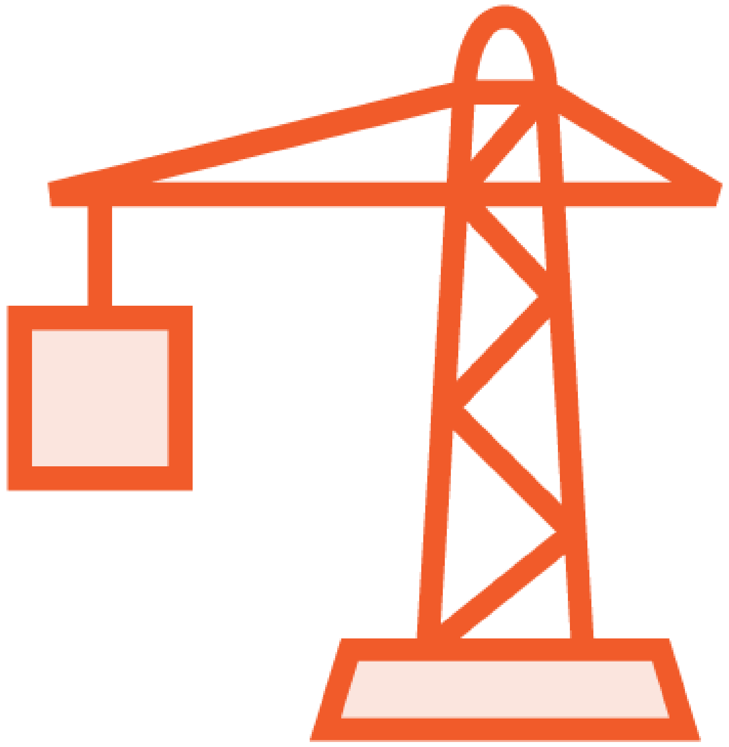
Spring Cloud detects binders on classpath

Can connect to multiple brokers of same type

Can also use different binders with same code

Possible to write your own binder

Explaining @StreamListener



Unique to Spring Cloud Stream

Handler for inbound messages

Does automatic content type conversion

Dispatch to multiple methods based on conditional checks

```
@EnableBinding(Source.class)
public class OrderSource {

 //auto push every 1 second
 @InboundChannelAdapter(
  value=Source.OUTPUT")
 public String sendOrder() {

  return “Polling Demo";

 }
}
```

- Lights up class as Stream app
- Source, Sink, Processor are built in, basic interfaces
- One way to emit data is with Spring Integration’s InboundChannelAdapter
- Return value of operation is sent to source output channel

```
spring.cloud.stream.bindings.
output.destination=orders

spring.rabbitmq.host=
  127.0.0.1
spring.rabbitmq.port=
  5672
spring.rabbitmq.username=
  rabbit
spring.rabbitmq.password=
  rabbit
```

- Properties or YAML file point to destination
- Destination name set here, or defaults to name of channel
- May also set connection values

```
@EnableBinding(Sink.class)
@SpringBootApplication
public class StreamReceiver {

 public static void
  main(String[] args)
 {
  SpringApplication.run(
   StreamReceiver.class,
   args);
 }

 @StreamListener(Sink.INPUT)
 public void log(String msg)
 {
  System.out.println(msg);
 }
}
```

◀ Lights up class as Stream app

◀ @StreamListener pulls from input channel of Sink

```
spring.cloud.stream.bindings.
input.destination=orders

spring.rabbitmq.host=
  127.0.0.1
spring.rabbitmq.port=
  5672
spring.rabbitmq.username=
  rabbit
spring.rabbitmq.password=
  rabbit
```

- Destination name needs to match value designated in source

Demo

Create new project via Spring Initializr

Add actuator and stream-rabbit dependencies

Create message sender in ”fast pass console” that publishes messages every second

Create message receiver in new project that processes streams of incoming messages

Observe RabbitMQ and what is automatically created

More Options for Producing Messages

Customize behavior of InboundChannel Adapter

Create custom interfaces for channel definitions

Push messages by injecting bound interface or channel directly

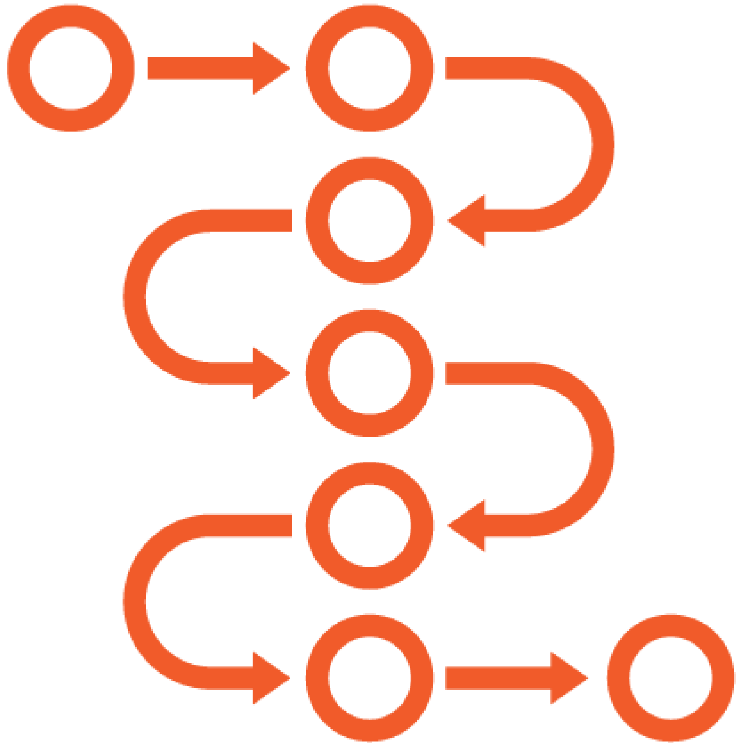
More Options for Consuming Messages

Use Spring Integration’s ServiceActivator

Set “condition” on @StreamListener to dispatch to different methods

Leverage @SendTo if consuming and re-publishing

The Role of Processors in Spring Cloud Stream



Stream app that sends AND receives

Has inbound and outbound channels

Use @SendTo to set output destination for the data returned by a method

Possible to use different broker instances, or types for the inbound/outbound channel

Demo

Create custom interface for sender application

Use InboundChannelAdapter to automatically publish two seconds

Use ServiceActivator on receiver

Create processor application

Dispatch messages to multiple StreamListener methods

RabbitMQ Binding Properties

Set up connection using spring.rabbitmq.*

Maps destination to a TopicExchange

Queue bound for each Consumer Group

Capable of handling retries (maxAttempts)

Can set routing key, point to existing queue, or change exchange type

Support for dead-letter queues

Control concurrency, header behavior, batching

Apache Kafka Binding Properties

APACHE kafka™ A distributed streaming platform

Setup broker, Zookeeper node references

Map destination to Kafka topic

Partitions, consumer groups map directly

Create new topics, or use existing ones

Define replication, rebalancing behavior

Control over offset handling

Using Consumer Groups to Scale

Use to scale up subscribers. Message goes to single instance in each group

Consumer group subscriptions are durable; when consumer group NOT specified, subscription is non-durable by default.

Set spring.cloud.stream.bindings.<channelName>.group property.

Demo

Add new REST endpoint to toll rate console app that sends to queue upon request (not polled)

Add consumer group property to receiver, observe change in RabbitMQ

Start multiple instances of toll intake receiver application and observe behavior

Stateful Processing with Partitions

Modeled after Apache Kafka behavior

Data split, processed by unique consumer instance

Useful for load balancing, stateful processing

Required properties for producer, consumer

Demo

Add partitioning based on toll station

Update receiver to store data for each toll station

Update sender to be aware of partition count

Start up receivers with arguments indicating the instance index

Submit messages and see which partition gets it

Working with Content Types

contentType header on outbound messages

Set declaratively via property setting

Native support for JSON, POJO, object, String, byte[] conversions

Use bean for custom message converters

@StreamListener automatically converts based on header

Spring Cloud Stream Health and Metrics

Monitor health of individual binders

Emit metrics to Spring Boot Actuator endpoint

Push metrics to streaming channel

Summary

Overview

Role of messaging in microservices

Problems with the status quo

Describing Spring Cloud Stream

Creating Stream applications

The role of processors

Using Consumer Groups to scale

Doing stateful processing with partitions

Working with content types

Health and monitoring