Microservice Architecture Blueprint

Episode 6 “Final Half of the Working Example“

Kim Horn

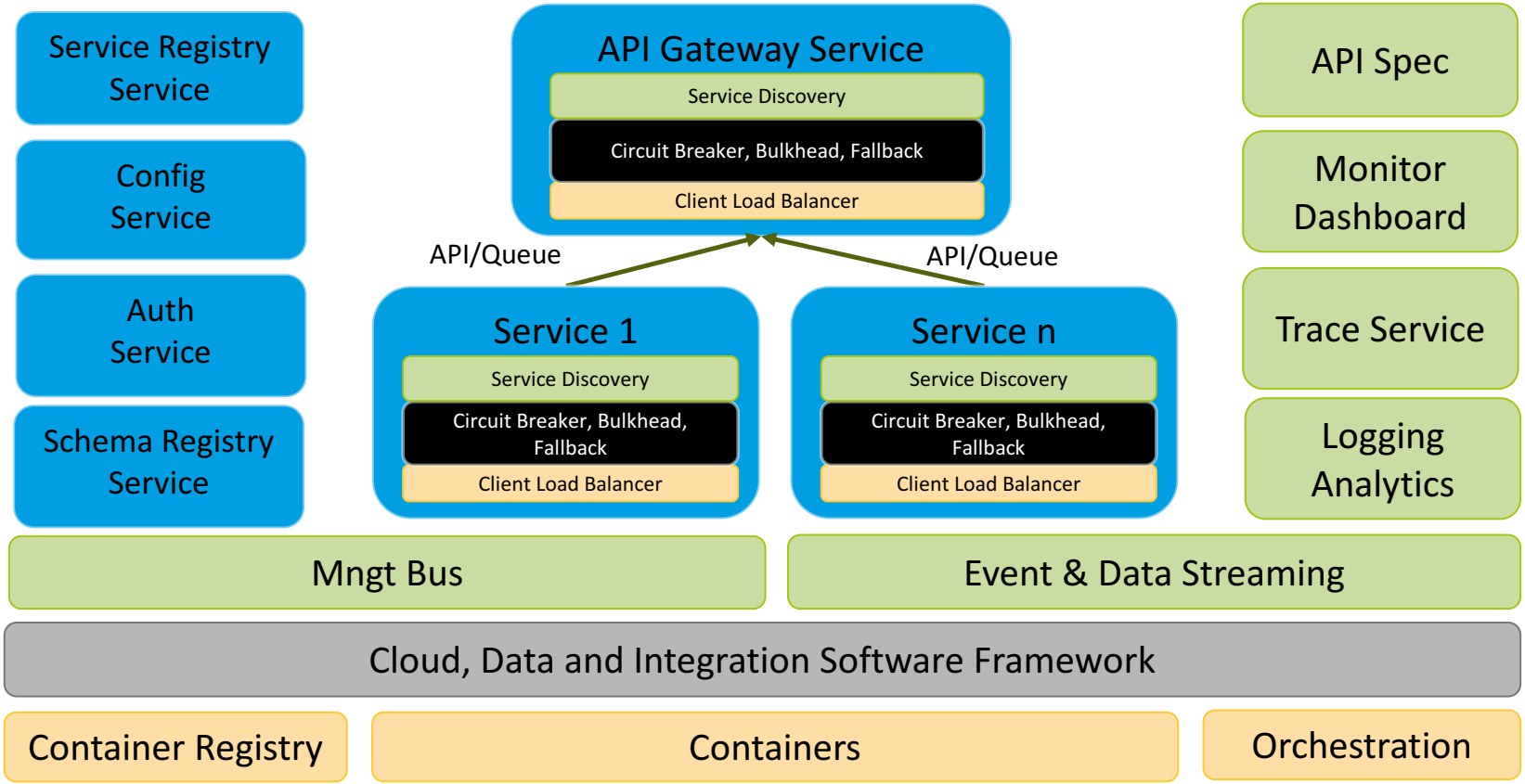
Version 1.151 **2 August 2017**

Agenda

- Legacy Systems – Wrap, Strangle Legacy and Replace ESBs;
- Event Driven Architecture (EDA) Style
- Streams: RabbitMQ and Kafka
- Tracing Through Asynchronous Streams- Kafka
- AWS SQS, SNS and WebHooks
- Orchestration Vs Choreography – Lightweight BPM
- DaaS - Lambda, Pipes and Filters Architecture Style - Data Streams

Working Blueprint

Distributed Cloud Blueprint – NFRs, use what you need



Common Tech For Patterns and Stack

Pattern / Capability	Tech
Gateway (reverse proxy)	Zuul 1&2 (Netflix) – wrapped by SpringCloud, Spring Cloud Gateway
Circuit Breaker, Bulkhead, Fallback	Hysterix (Netflix) – wrapped by SpringCloud
Client Side Load Balancer	Ribbon (Netflix) – wrapped by SpringCloud
Service Discovery Registry	Eureka (Netflix) – wrapped by SpringCloud / Consul
REST HTTP and Service Discovery Client	Feign (Netflix) – wrapped by SpringCloud
Configuration Service	Spring Cloud Configuration
Authorisation Service	Spring Cloud Security, SAML, OATH, OpenID
Event Driven Architecture, Mngt Bus	Spring Cloud Stream wraps Kafka (also supports Rabbit MQ), Spring Cloud Bus
Data Analysis and Flow Pipelines	Spring Cloud Data, Spring Cloud Data Flow, Sparks
Core Software Cloud Frameworks	Spring Boot, Spring Actuator, Spring Cloud, Spring Data, Spring Integration
IaaS, Container, Orchestration	Docker, Compose [Kubernites, Swarm]
Logging, Tracing & Analysis	SL4J, ELK[Elastisearch, Logstash, Kibana], Sprint Cloud Sleuth, Spring Cloud Zipkin.
In memory Cache	Spring Data Redis
Analytics, Monitoring Dashboard, Orchestration(bpm)	Atlas, Turbine, Visceral, Camuda, Conductor (lightweight)
API/ Service Specs, Schema Registry,Testing	Spring Hateoas (HAL), Spring Fox (Swagger), Avro, Protobuf, Spring Cloud Contract, WireMock

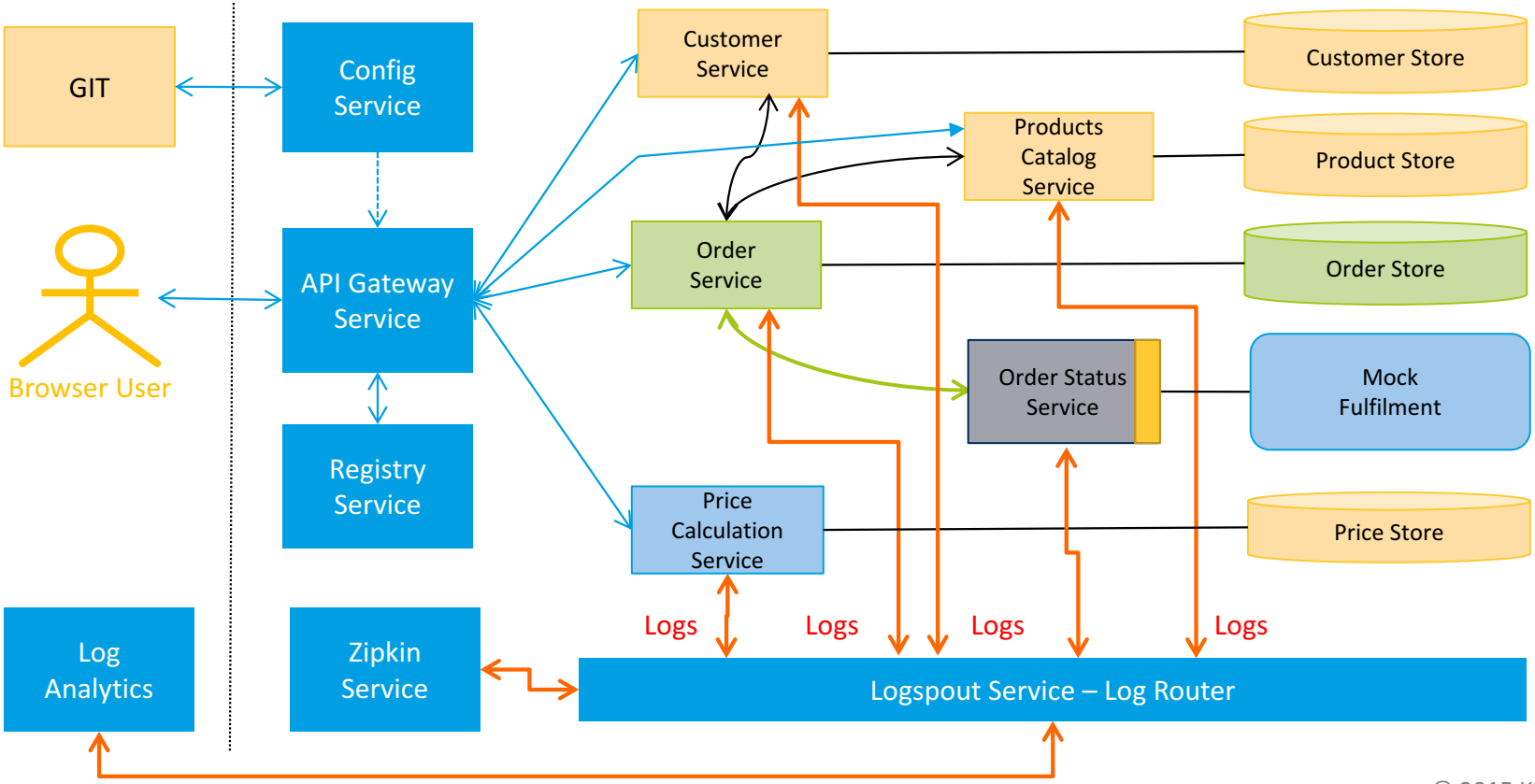
Microservice Principles

A Microservice is:

1. Modelled around a bounded context; the business domain.
2. Responsible for a single capability;
3. Based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven;
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small autonomous team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

RECAP - 12 Services

RP



Legacy Systems

Legacy Systems

- How can Microservices be used to integrate with Brownfields and Legacy systems, or systems that do not expose APIs or Microservices?
- How can we replace Monolithic ESBs that provide SOA integration, that require large complex builds, to allow for Continuous Deployment approaches?

Solutions:

- Wrapper Pattern
- Strangler Pattern

Wrapper Pattern

- Use Wrappers to insulate and protect components from change.
  - Use Adapters to integrate heterogeneous back end systems;
  - This also add an Anti Corruption Layer.
- Expose Components with a Façade - Interface that allows the Implementation or backend of the component to be changed.
- Implementation can be changed at will without impacting consumer.

Strangler Pattern

The strangler pattern is used where a legacy system cannot be totally re-built or opened with APIs and so a feature is strangled with new code to open it to integration.

- Make something new that obsoletes a small percentage of something old.
- Put them live together.
- Repeat.

This is a good pattern to prevent feature branches in source control with all of the integration headaches that go with it.

- No one forced to use the new systems;
- The old system may never be removed;
- Can move between new and old.

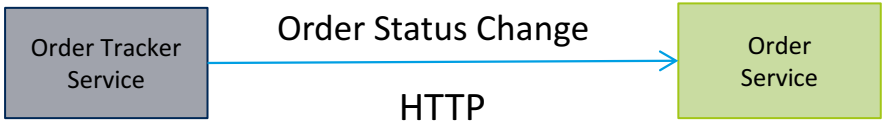
Strangler Steps

The Strangler Application steps are :

- **Transform**— Create a parallel new service, even in your existing environment but based on Microservices, that covers a small bounded domain (feature);
- **Coexist**— Leave the existing service where it is for a time. Redirect from the existing service to the new one so the functionality is implemented incrementally;
  - Use API gateway to redirect traffic;
- **Eliminate**— Remove the old functionality from the existing site (or simply stop maintaining it) as traffic is redirected away from that portion of the old site.

Event Driven Architecture Style

Order Update - Status + Activation Tracking



- Having the Order Tracking Service service call the Order Service to update the Order Status would introduce tight coupling.
- The coupling has introduced brittleness between the services. If the Order service endpoint changes, the Order Tracking Service has to change.
- The approach is inflexible because we cannot add new consumers of the Status Tracker and Activation Service without modifying the code on theses services to know that they have call the new services to let them know about the changes.
- The HTTP call is synchronous and unrelaible.

Asynchronous Status Update Messages

- The solution is to decouple these service using messages.
- The order service monitors a Queue to see if there are any status updates.
- This approach offers four benefits:
  1. **Loose Coupling** – consumers and providers have no knowledge of each other;
  2. **Durability** – if the consumer service is down the provider just does its business as usual, messages will eventually get consumed when it becomes available;
  3. **Scalability** – provider does not need to wait for consumer to read message, if not enough consumers to process messages more can be added;
  4. **Flexibility** – new types of consumers can be added without the provider needing to know what the message is used for.

Distributed Sessions and State.

- **Problem**: consistency, reliability and scalability; by having session state you make one, or both difficult.
- **Problem**: business process management requires management of process (application) state. If this is done on the server side, difficult to scale, as state has to be distributed consistently across servers. Storing state to a DB is expensive.
- A distributed cache may span multiple servers so that it can grow in size and in transactional capacity. It is mainly used to store application data residing in database and web session data.

Types of Events

- **Notifications** – used to reverse a dependency (e.g. from an RPC); we don't care who gets the message but something has happened, e.g. 'Order status has changed';
  - **Commands** – one system wants another to do something explicit, e.g. 'Update Order Status to XXX';
  - **Broadcasting** – multiple consumers listening;
  - The data itself (resource representation) is not sent but the type of change;
- **Messaging** (called Event Carried State Transfer by M Fowler)
  - send all the data required (the representation of the resource) for the dependent system to carry out their job, change their state, without going back to sender;
- **Event Sourcing** – ability to re-create history of changes to get final state;
  - version control for data;
  - To get to now just replay the stream of events.

Tradeoffs.

- **Notifications** make it very easy to add new consumers, as the sender does not need to know anything;
  - **But** make systems hard to understand.
    - If systems just send events they are decoupled, but it can get very difficult to see what the whole system does.
    - Can't look at the 'program code' to see all the consumers and senders of messages and the process flow is lost.
- **Messaging**
  - improves availability, reduces network calls, but then requires data duplication, receiver has to have all the data it requires from sender. Thus consistency issues - BASE;
  - Large data sets that are not essential can cause bandwidth issues;
  - Sending whole resource when only a single attribute required.
- **Event Sourcing** – Asynchrony and Versioning are difficult; what event do you store, Input, Internal, Output;

Event Driven Architecture Style

An Event Driven Architecture is a style that focuses on the production, detection and consumption of events. Events are happenings of interest, typically changes in state that are externalised for other processes, consumers, to action.

A typical Event Driven sequence is:

1. A functional process will Produce an event based on an outcome;
2. The event is then published over a Channel;
3. Consumers that have registered to receive this event listen on that Channel, then consume the event;
4. The consumer may or may not process it. They may call the producer service directly to get the resource of interest;
   - Simple (micro) events may include the resource in the message;
5. A stream - events in time order follow a business process.

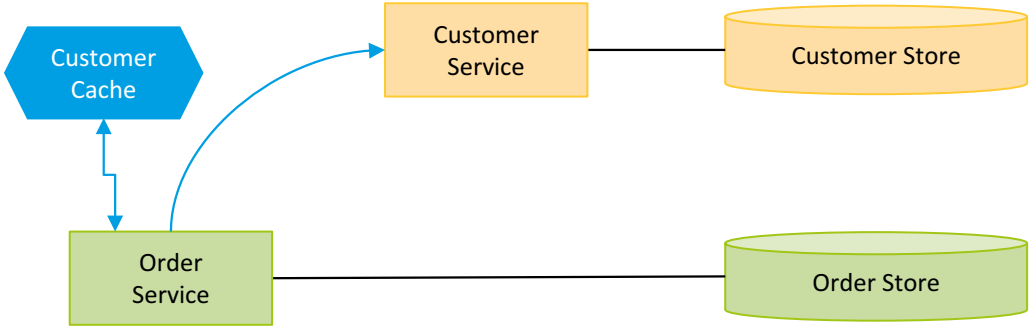
Event Driven Architecture Recomendation

Events should typically contain only minimal information sufficient for the consumer to then retrieve the resource required and so the payload is not large.

- Consumers and producers may listen to and produce events on different channels;
- An important feature of event driven is that the source system does not care about a response, or if it does it does its indirect, and there is a distinct separation between the sending and receiving logic;
- Give the consumer the choice to respond to event and get the required resource data;
- Event Driven styles utilise Message Driven infrastructures and a mechanism such as publish-subscribe;
- Producers should be localised to a Bounded Context;
- Producers and consumers are completely decoupled. A producer has no knowledge of the consumer.

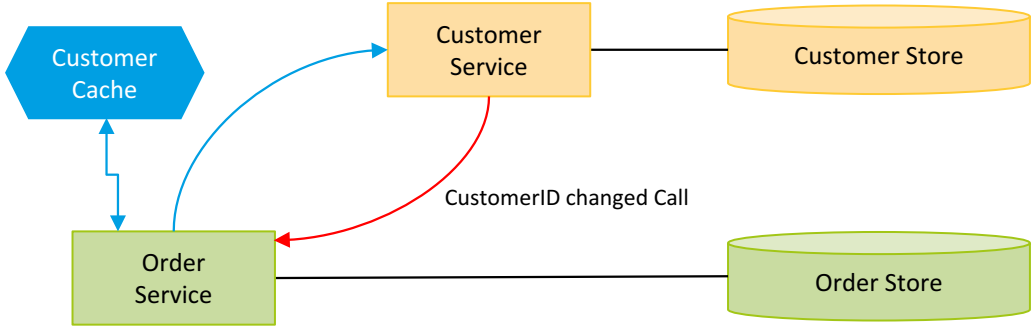
Caching Update Problem

Cache Customers in the Order Service to speed up lookups.
Problem is what to do when Customer is updated. No way to tell the Cache.



Customer Service can call the Order Service, and pass the changed Customer ID. Order service exposes a new API for this.

However this introduces another dependency, making the system brittle. Not recommended.



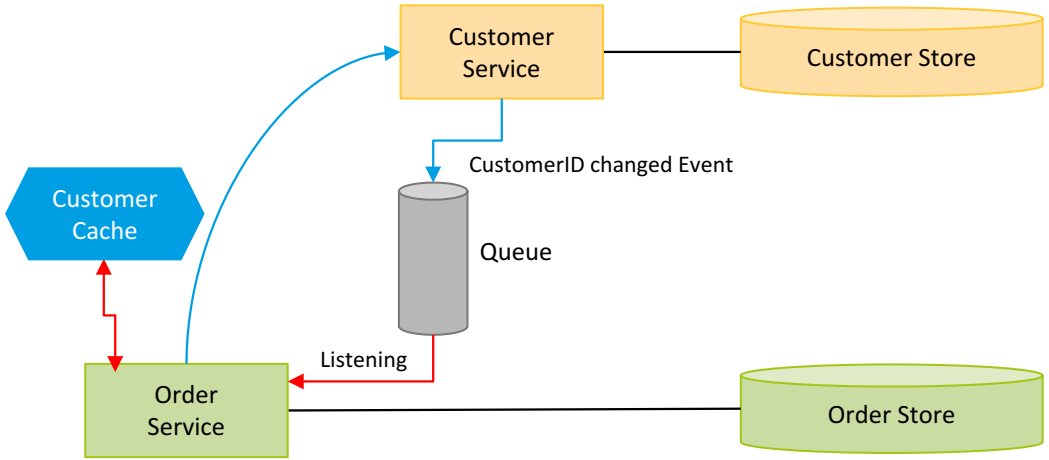
Solution – Add a Queue (Message Broker)

Employ a Queue, and publish an event notification when a customer is changed.

Order service listens to the Queue and invalidates the cache with a message that the Customer has changed.

The customer is read again, via usual HTTP API, and cached.

The Queue decouples the 2 services. The Order service only knows about the Queue, and nothing about the Order service.



Could be a Queue, like Rabbit MQ or a notification service (e.g. AWS SNS) , or message stream, like Kafa, or other message broker.

Solution – Add New Consumers any time

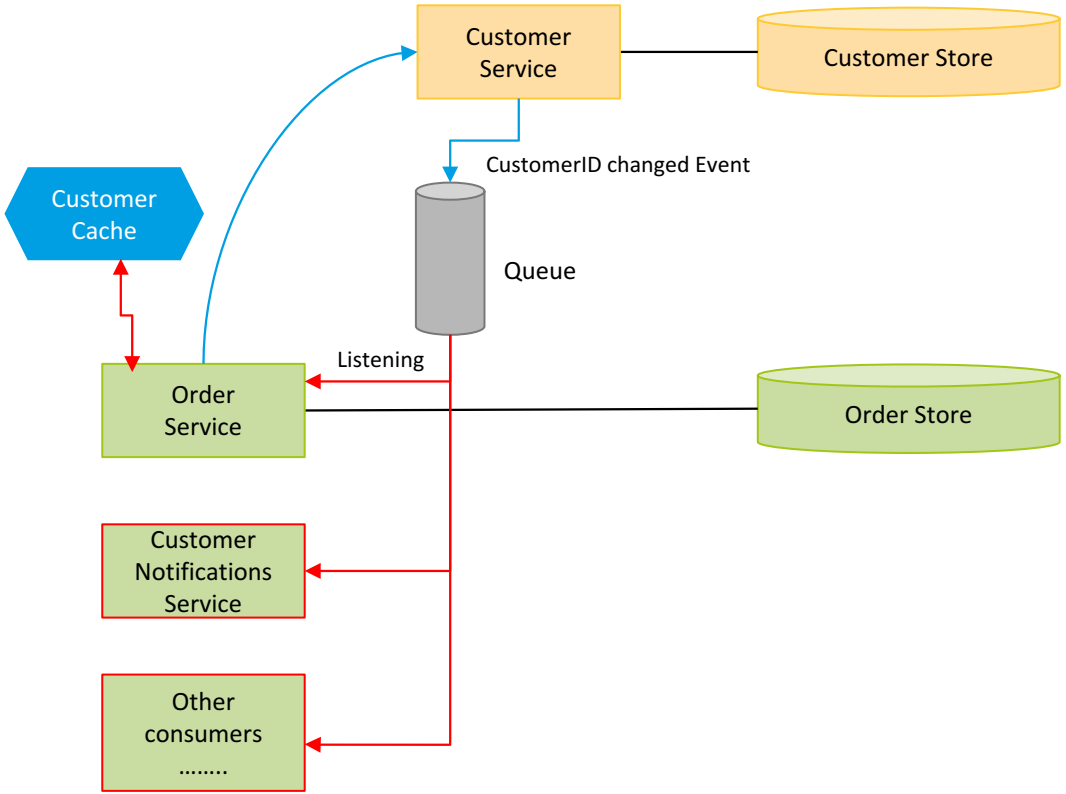
New Service can be added as consumers of the Change easily.

For example a service that notifies the customer when changes to its details are made.

Other consumers can be added , without any change to Customer service.

The Customer service does not need to know about any of the new consumers.

Provides excellent decoupling



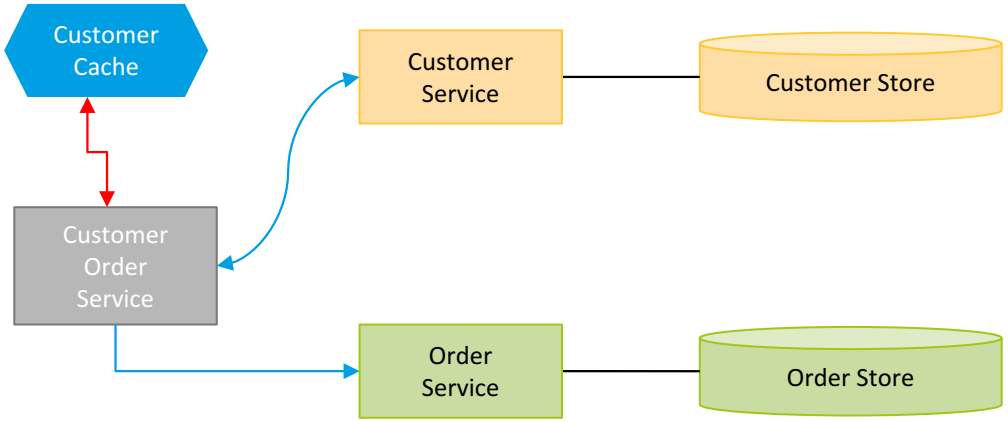
Alternate Solution – Proxy Customer and Order – Not so Good

Employ a Customer Order Service that proxies the customer service (has all customer APIs) and provides Customer Orders.

Changes to the Customer can be directly updated in the cache.

This is less attractive as customer service API changes are coupled to Customer Order Service.

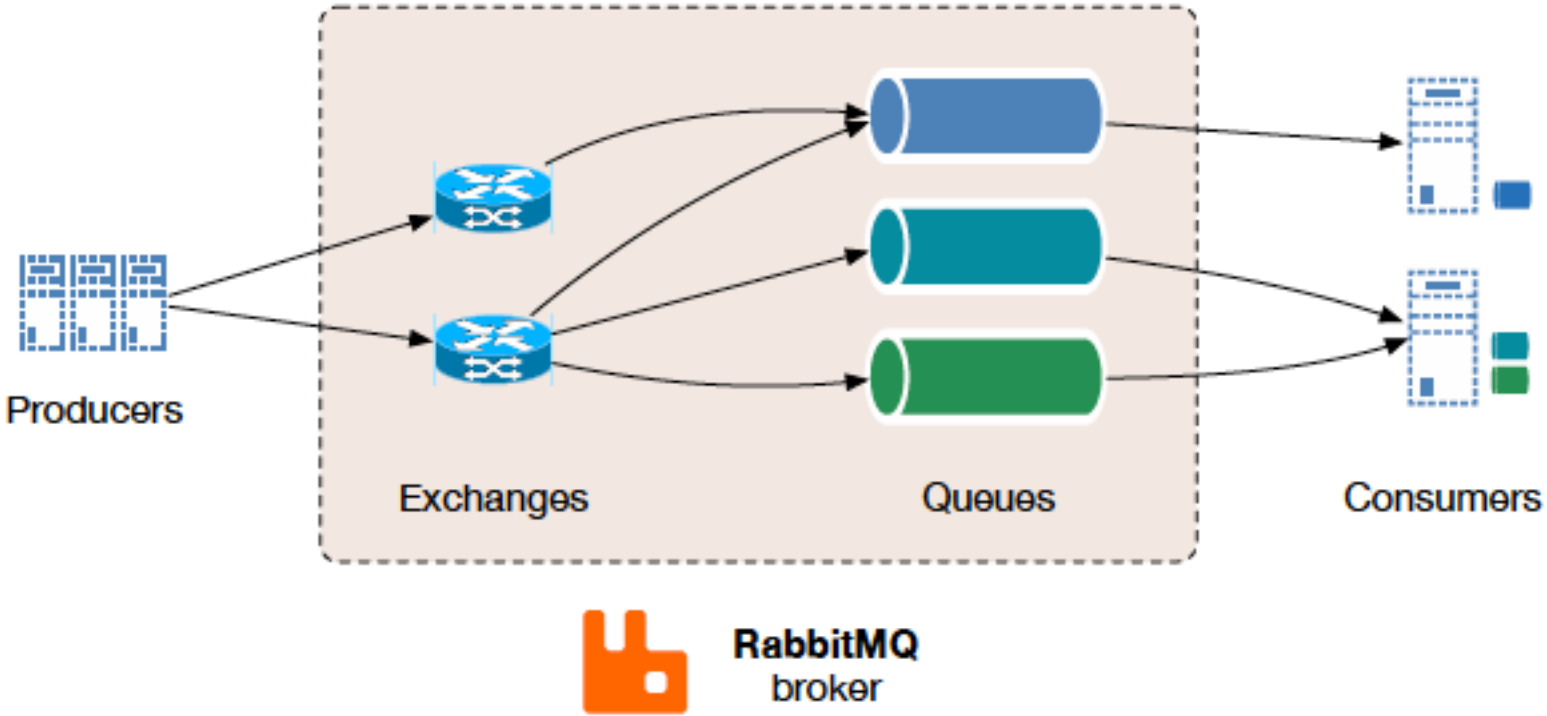
Other Consumers of Customer will have to do the same, and so this creates a complex network of tightly coupled services



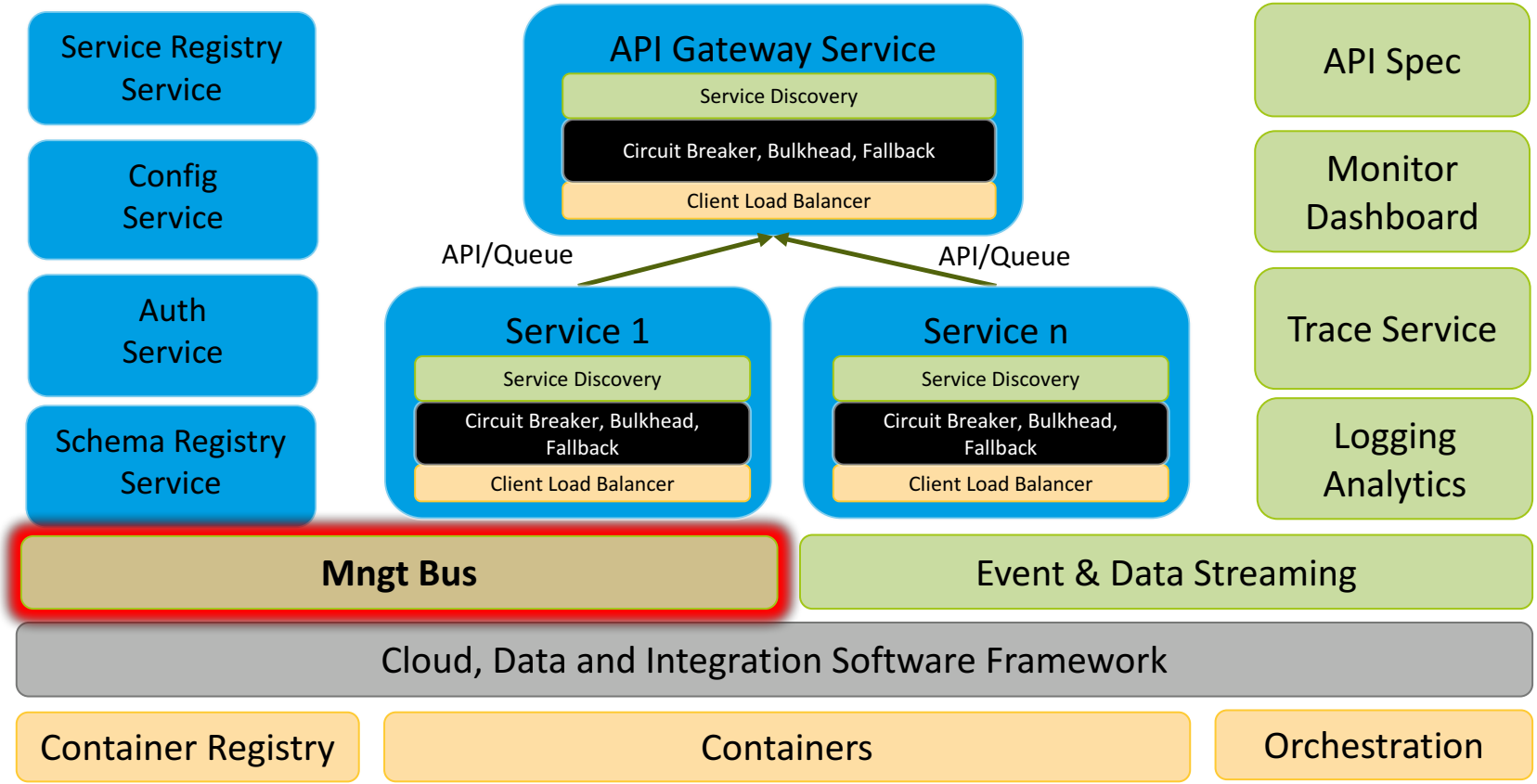
RabbitMQ & Kafka

- Designed as a general purpose message broker, sync or async, employing several variations of communication styles patterns:
  - point to point,
  - request/reply and
  - pub-sub
- **It uses a smart broker / dumb consumer model**, focused on consistent delivery of messages to consumers that consume at a roughly similar pace as the broker keeps track of consumer state.
- It is mature, performs well.
- Built to support AMQP (Advanced Message Queue Protocol) and any combination of protocols like AMQP 0-9-1, STOMP, MQTT, AMQP 1.0.
- Often used with Apache Cassandra when application needs access to stream history.
- Offers a number of distributed deployment scenarios;
- Complex routing to consumers, integrate multiple services/apps with non-trivial routing logic;
- 20K messages per second is easy to push through a single Rabbit queue

RabbitMQ

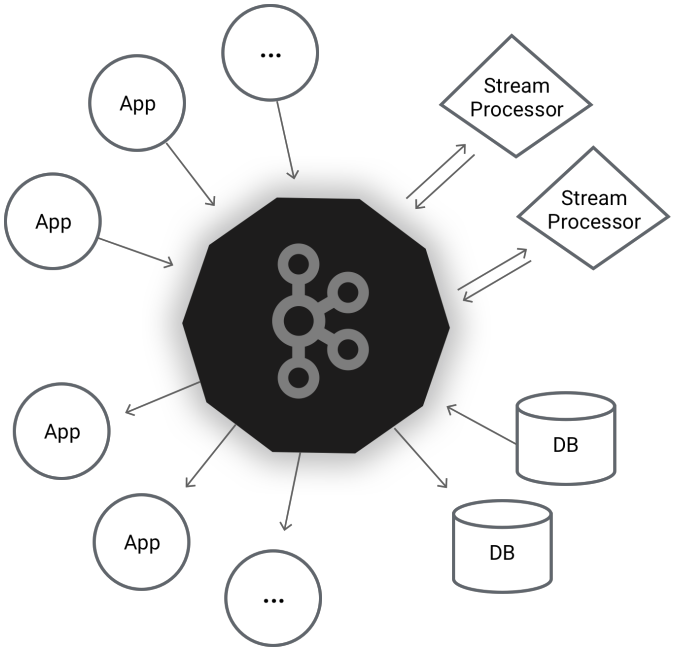


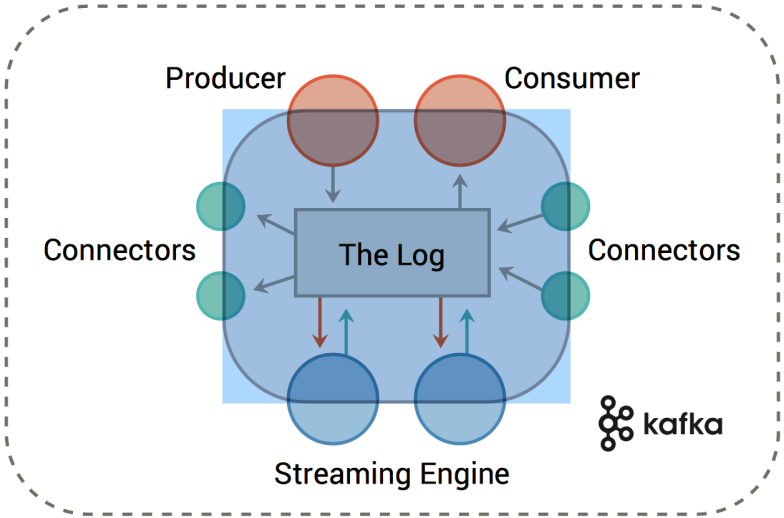
Distributed Cloud Blueprint – Rabbit often used as Mngt Bus



Kafka Platform

Apache Kafka™ is *a distributed streaming platform*. Provides decoupling, reliability and scalability.





"ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications."

https://zookeeper.apache.org

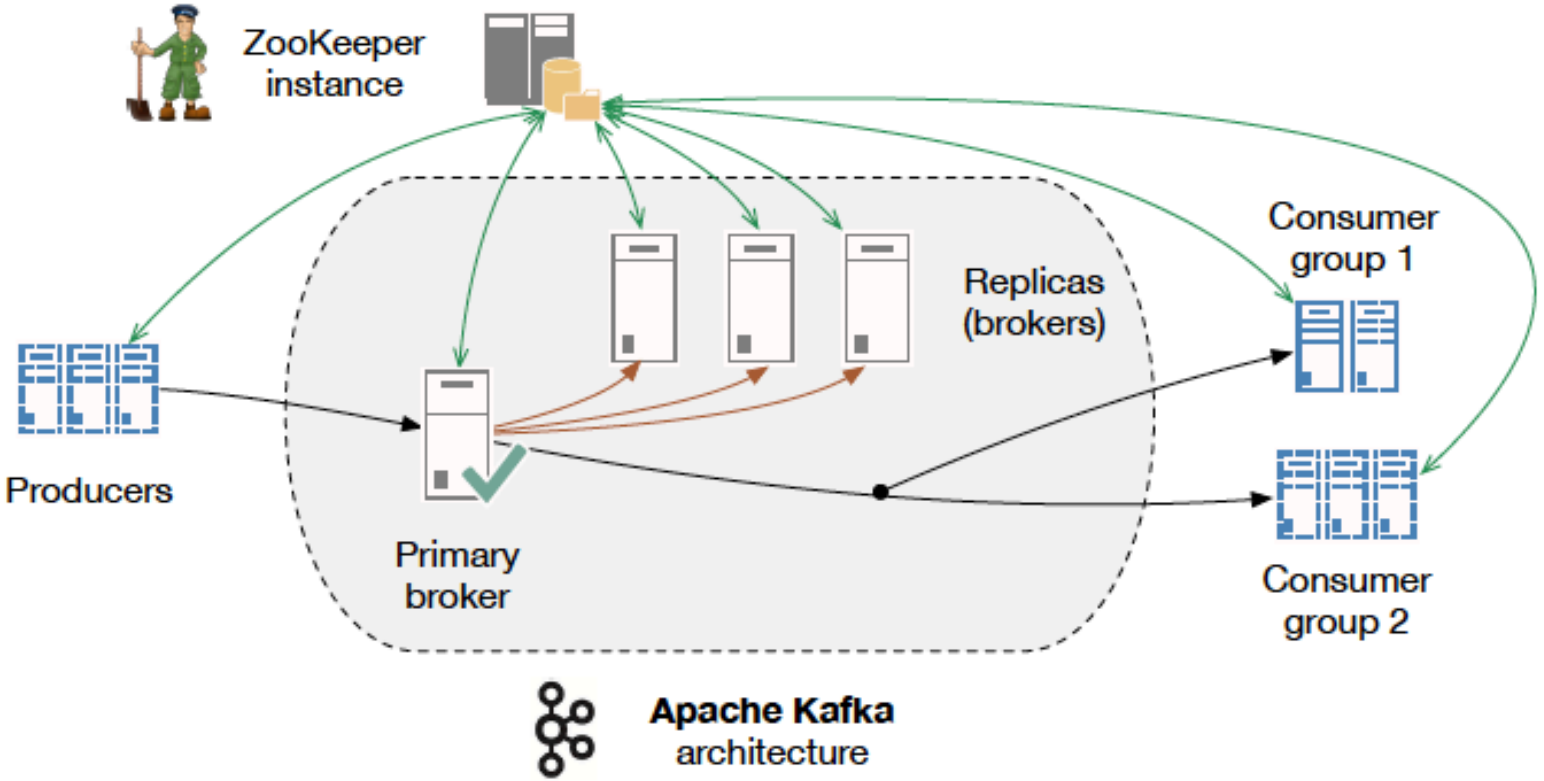
Kafka and Zookeeper

Kafka uses Zookeeper for:

- The election process for the controller broker. If the controlling broker fails or becomes unavailable, Zookeeper elects a new controller from a set of brokers that are considered to be caught up with the leader. The controller broker is responsible for setting up leader/follower relationships for all partitions of a topic.
- Cluster Membership - Joining the cluster and maintaining membership in the cluster. If a broker becomes unavailable, Zookeeper removes the broker from cluster membership.
- Topic configuration - what are the topics in the cluster, which broker is the leader for a topic, what are the number of partitions for a topic and what are the specific configuration overrides for a given topic.
- Access Control - who can read and write from particular topics.

Kafka

- Designed for high volume publish-subscribe messages and streams, meant to be durable, fast, and scalable.
- Provides a durable message store, similar to a log, run in a server cluster, that stores streams of records in categories called topics.
- Every message consists of a key, a value, and a timestamp.
- Nearly the opposite of RabbitMQ, Kafka employs **a dumb broker and uses smart consumers** to read its buffer.
- Does not attempt to track which messages were read by each consumer but retains all messages for a set amount of time, and consumers are responsible to track their location in each log (consumer state).
- Stream from A to B without complex routing, with maximal throughput (100k/sec+), delivered in partitioned order at least once.
- Has recently added Kafka Streams which positions itself as an alternative to streaming platforms such as Apache Spark, Apache Flink, Apache Beam/Google Cloud Data Flow.



Kafka Use Cases

- **Website Activity Tracking** - original use case was to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds.
- **Messaging** - a replacement for a more traditional message broker.
- **Metrics** - operational monitoring data.
- **Log Aggregation** - as a replacement for a log aggregation solution.
- **Stream Processing** - process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing
- **Event Sourcing** - a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.
- **Commit Log** - serve as a kind of external commit-log for a distributed system.

Spring Cloud Stream

Spring Cloud Stream:

- make it significantly easier to use Kafka and RabbitMQ.
- Is a framework for building message-driven microservices based applications.
- Augments Spring Boot to create DevOps friendly stream applications.
- Augments Spring Integration to provide connectivity to message brokers.
- Provides an opinionated configuration of message brokers, introducing the concepts of persistent pub/sub semantics, consumer groups and partitions across several middleware vendors.

Spring Cloud Stream

By adding @EnableBinding to your main application, you get immediate connectivity to a message broker.

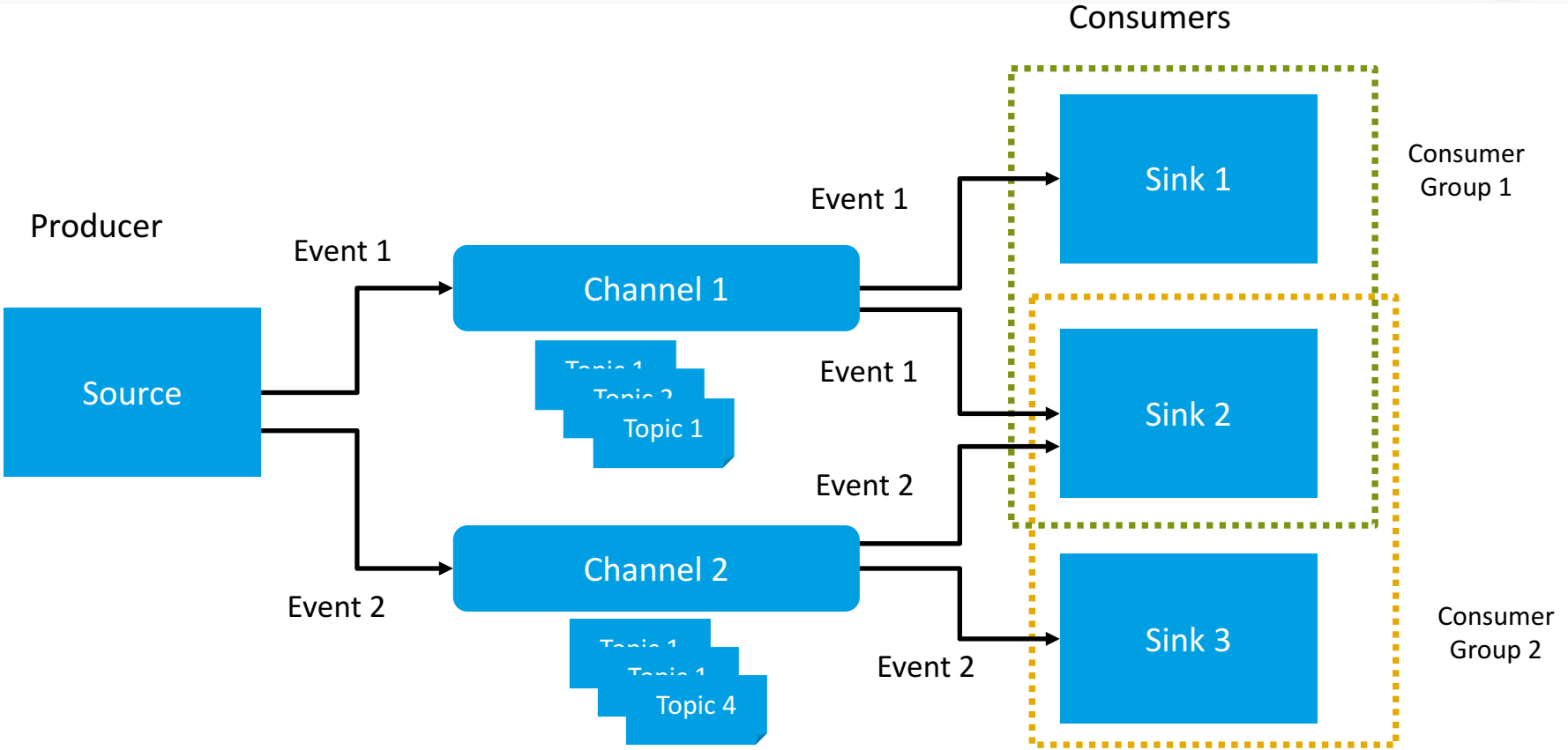
By adding @StreamListener to a method, you will receive events for stream processing.

Transports:

- Kafka
- RabbitMQ

Need to review requirements carefully, although Kafka is trendy and shiny, it may not be better than Rabbit, in your case. Rabbit is far easier to set up and requires significantly less infrastructure than Kafka (See separate pack comparing the 2)

Source, Sink, Channel, Topics

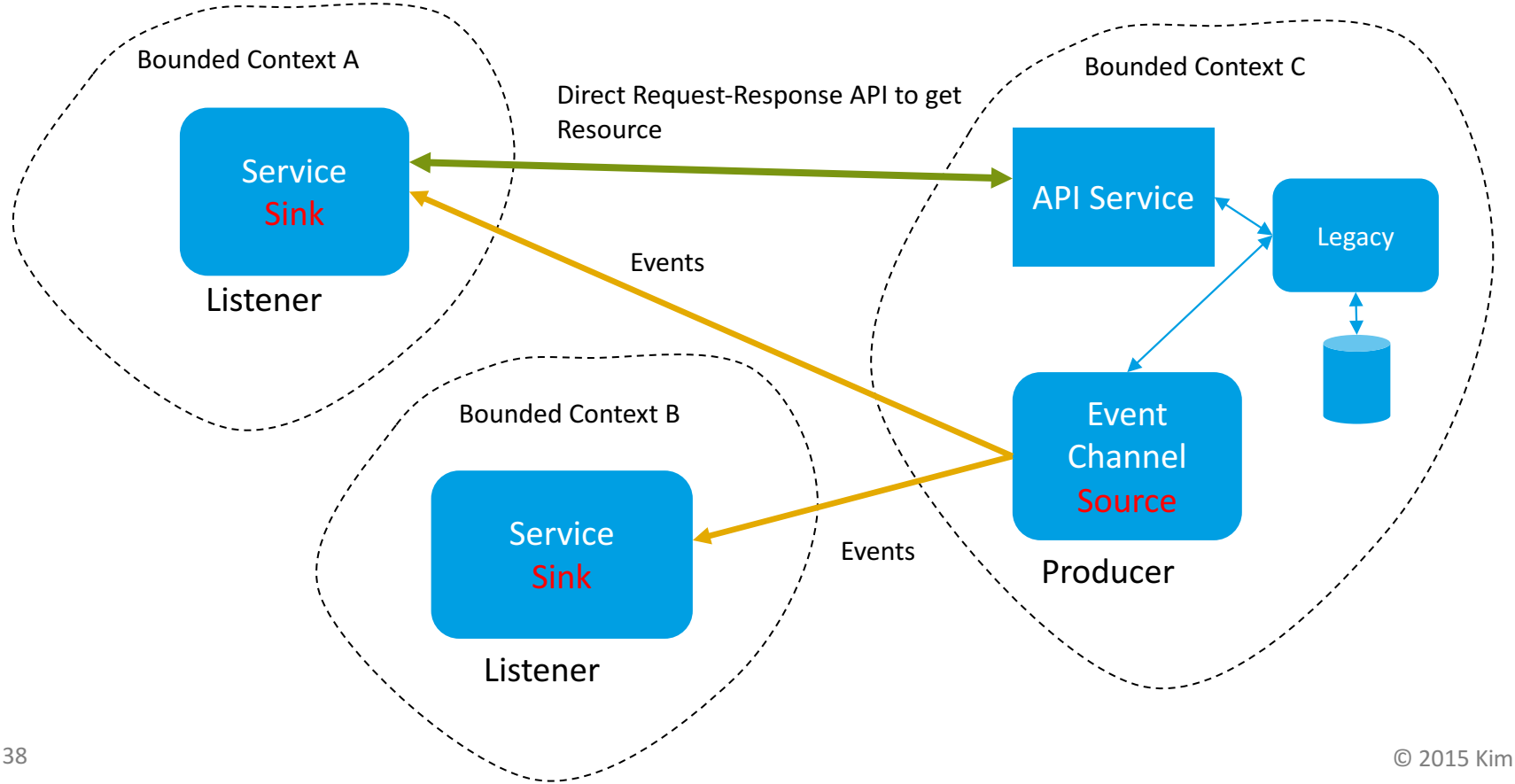


Consumer Groups

Kafka Broadcast:

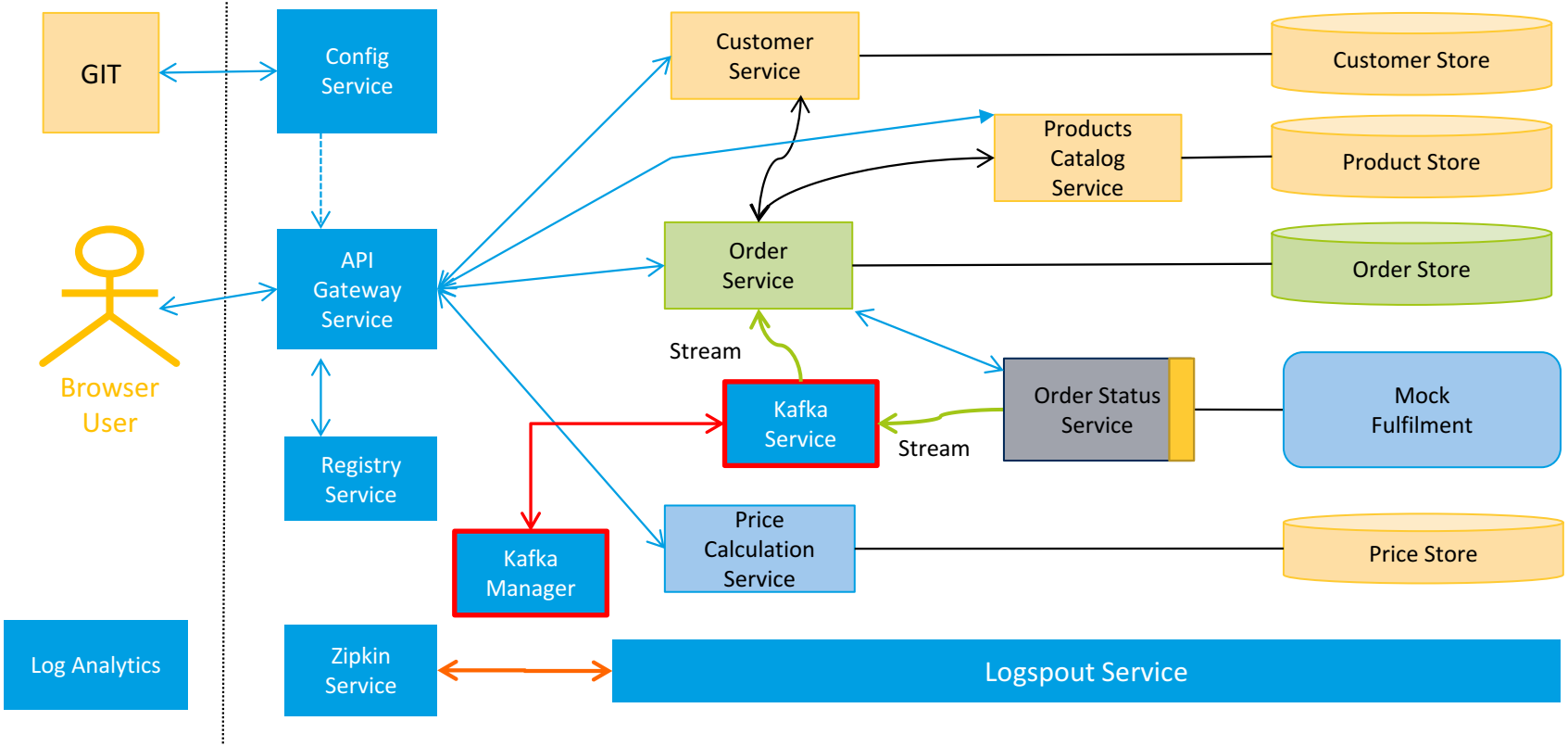
- Only one consumer in a consumer group can pull a message. But **all** consumer groups get the messages.
- If you want all your consumers to get the messages, assign them different consumer groups. Each message goes to every consumer group, but within a group, it goes to only one consumer.

Service Pattern with Events



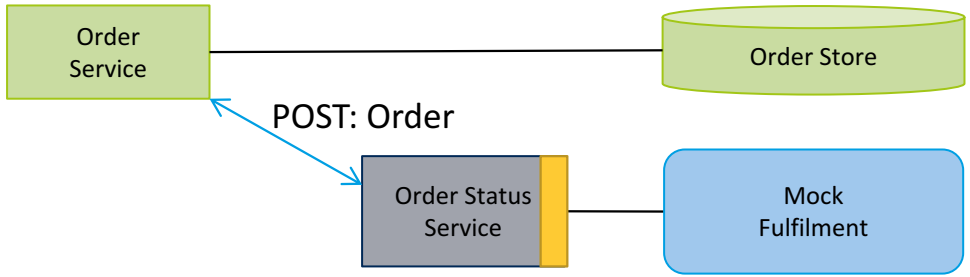
Stream Service - 14 Services

RP



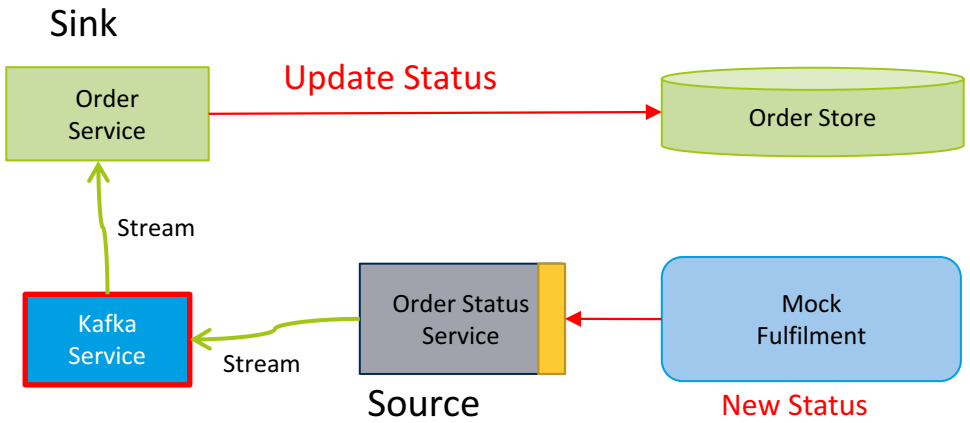
Order Submission to Fulfilment

Step 1: The order is 'Submitted', POST, and sent to Fulfilment via Order Status Service. This Service Wraps the Legacy Fulfilment System



Step 2: Each time the Status of the Order changes in Fullfilment a new Event is produced on a Channel.

- The Order Service listens to this channel and updates the Order Store.



Event Sourcing Approach

- Use a service with a data base to record the history of events;
- The current state is the latest event;
- The events can be replayed at will, to recreate the current state;

Date	Event
1/2/2001	-$10
2/3/2001	+$40
4/3/2001	-$3
10/4/2001	+$15

Balance at end 2000 = $100

At end of March = $127
At end of May = $142

Order Status - Event Sourcing

In the example the Order Status Service will record the status changes for the order as events are received from Fulfilment

#	status_id	order_id	status
1	f354f8c-c443-4ebe-a82a-e2fc1d1ff78a	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	received
2	5b6d2ad8-bc6a-4dd3-9279-ffa8cd36d55c	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	submit
3	c670712d-a25d-436e-93b1-aeaf450e68f0	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	acknowledged
4	b2da944b-066b-4f87-a028-fa940a214cb3	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	inprogress
5	69558253-bc81-4df7-a1ce-6a58d57ac2b6	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	completed
6	b75e21d0-5ff0-4aa4-881e-2affa971a026	f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a	activated

Kafka Streams and Tables

A stream is a sequence of records. Kafka models a stream as a log, that is, a never-ending sequence of key/value pairs:

- key1 => value1
- key2 => value2
- key1 => value3

A table is something like this:

key1	value3
key2	value2

So tables are just a particular view on a stream. Pat Helland said,
"a table is just a cache of the latest value for each key in a stream".

The log can be

- compacted once read,
- Rewound and read again

Tracing Through Kafka

Sleuth Across Kafka Topics

nbn_source | 2017-07-27 07:15:12.070 DEBUG [sourceservice,17202eec1ee0f340,17202eec1ee0f340,true] 12 --- [p-nio-80-exec-2] c.widget.events.source.SimpleSourceBean : Preparing Kafka message HOME

nbn_source | 2017-07-27 07:15:12.088 DEBUG [sourceservice,17202eec1ee0f340,17202eec1ee0f340,true] 12 --- [p-nio-80-exec-2] c.widget.events.source.SimpleSourceBean : Sent Kafka message HOME

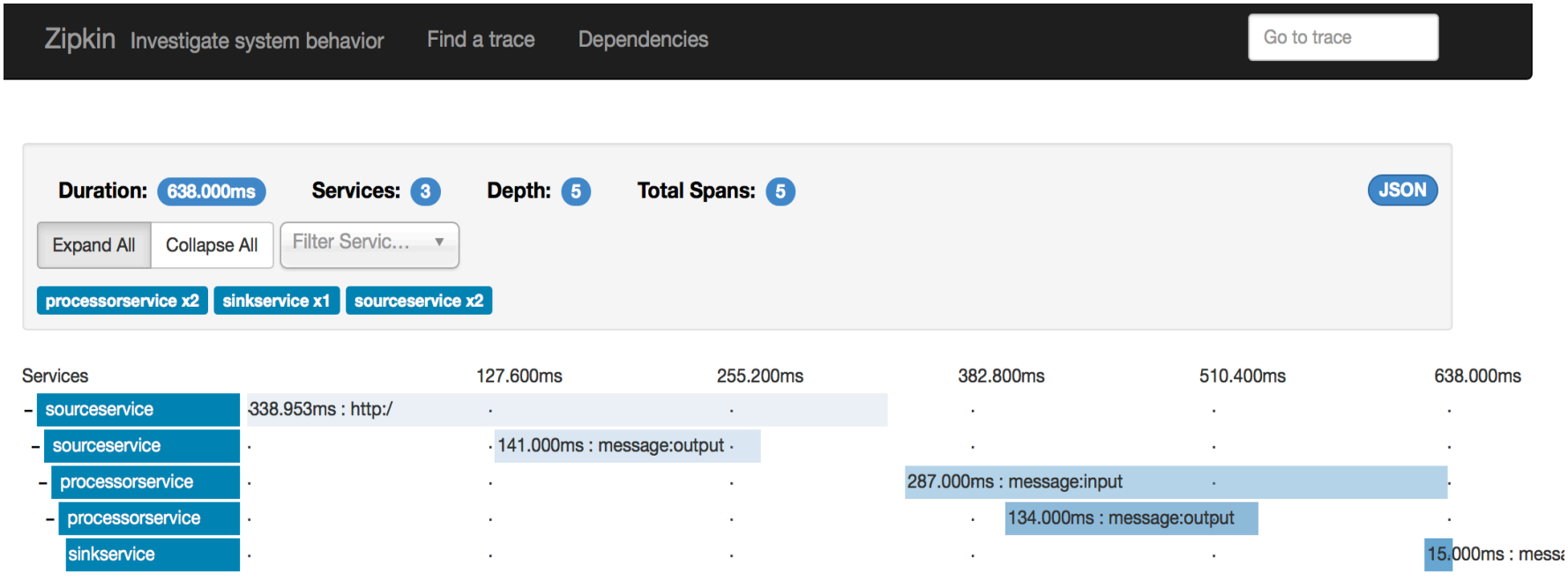
nbn_processor | 2017-07-27 07:15:12.100 DEBUG [processorservice,17202eec1ee0f340,b138ebb212d2276e,true] 14 --- [afka-listener-1] com.widget.process.Application : Transform In event Type: com.widget.event.models.MessageModel Action: HOME

nbn_processor | 2017-07-27 07:15:12.100 DEBUG [processorservice,17202eec1ee0f340,b138ebb212d2276e,true] 14 --- [afka-listener-1] com.widget.process.Application : Transform Out event Type: com.widget.event.models.MessageModel Action: TransfromHOME

nbn_sink | 2017-07-27 07:15:12.163 DEBUG [sinkservice,17202eec1ee0f340,6fb6d0c95ccbfd17,true] 13 --- [afka-listener-1] com.widget.sink.Application : In SINK Method

nbn_sink | 2017-07-27 07:15:12.163 DEBUG [sinkservice,17202eec1ee0f340,6fb6d0c95ccbfd17,true] 13 --- [afka-listener-1] com.widget.sink.Application : Received an event Type: com.widget.event.models.MessageModel Action: TransfromHOME

Kafka Topic Traces via Zipkin



Configuration – Add Headers

```yaml
spring:
  cloud:
    stream:
      default:
        contentType: application/json
      kafka:
        binder:
          brokers: server1
          defaultBrokerPort: 9092
          zkNodes: server1
          defaultZkPort: 2181
          headers:
            - X-B3-TraceId
            - X-B3-SpanId
            - X-B3-Sampled
            - X-B3-ParentSpanId
            - X-Span-Name
            - X-Process-Id
  sleuth:
    enabled: true
    sampler:
      percentage: 0.5
```

When use Spring Cloud Stream with RabbitMQ the Sleuth info is handled automatically, however, with Kafka you need to add Headers explicitly

Spring Cloud Steam + Actuator Channels

http://localhost:8082/channels


```
{
    "inputs": {
        "input": {
            "destination": "messageSourceTopic",
            "group": "sourceGroup",
            "contentType": "application/json"
        }
    },
    "outputs": {
        "output": {
            "destination": "messageSinkTopic",
            "contentType": "application/json"
        }
    }
}
```

SQS, SNS, Webhooks

Amazon Simple Queue Service (SQS) and Notification Service (SNS)

SQS is a fully managed message queuing service

- SQS offers two types of queues:
  - *Standard queues* offer maximum throughput, best-effort ordering, and at-least-once delivery.
  - *FIFO queues* are designed to guarantee that messages are processed exactly once, in the exact order that they are sent, with limited throughput.
- Multiple copies of every message is stored redundantly across multiple availability zones so that they are available whenever applications need them.
- Four basic APIs: CreateQueue, SendMessage, ReceiveMessage, and DeleteMessage.
- message payload can contain up to 256KB of text;
- Combined with Amazon Simple Notification Service (SNS), messaged can 'fanout' to multiple standard SQS queues.
- When a message is received, it becomes “locked” while being processed. This keeps other computers from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again.

WebHooks - Jeff Lindsay in 2007

WebHooks are user-defined HTTP callbacks; a way of augmenting or altering the behavior of a web page, or web application, with custom callbacks.

- Event Driven, and when that event occurs, the source site makes an HTTP request to the user configured URI.
- Maintained, modified, and managed by third-party users and developers who may not necessarily be affiliated with the originating website or application.
- Have been used to build a message queuing service on top of HTTP. RESTful examples include IronMQ and RestMQ.
- Issues:
  - Do not scale well compared to using Queue/Kafka.
  - Requires a fair amount of work to get right given network issues.
  - Create dependencies, i.e. call back is shared contract

Orchestration Vs Choreography

Managing Orchestrations and Workflow

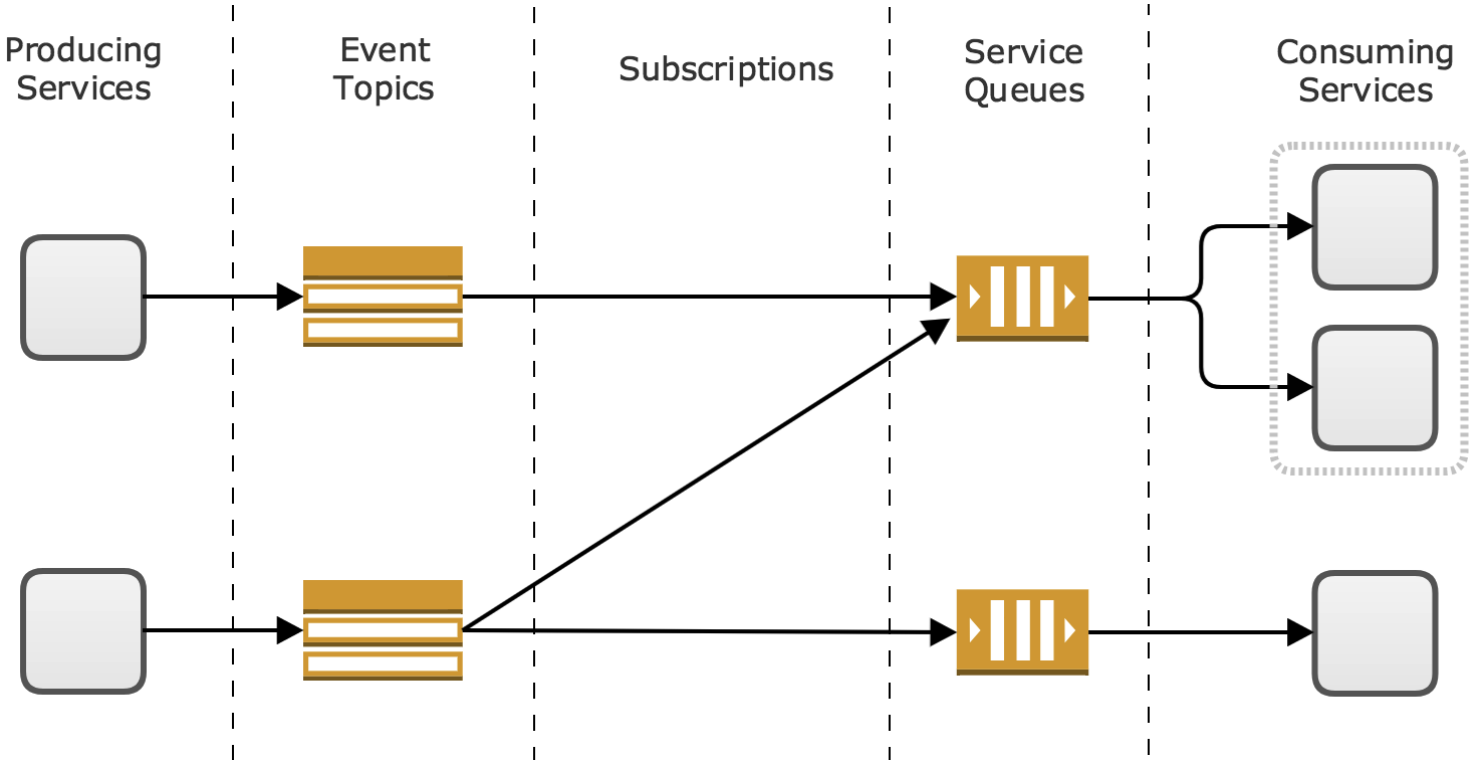
You can build specific microservices that are business workflow services, that manage workflow state, using a data base.

- It can orchestrate other microservices directly via REST calls or Asynchronously via events.
- It can wait for services to respond, or wait for events.
- It can respond to distributed events, e.g. work completed, via queues.
- It can provide an API to allow state to be queried.

Choreography is an alternate approach where the ‘process’ is distributed out to all those involved.

- It gets difficult to know what the actual process is, when no single ‘engine’ knows.
- It is very difficult to make available the process or workflow description, built into an add hoc microservice. Without this knowing the state is not much use even if it is externalised.
- At some stage as the number of workflows to be managed increases, and things get more complex a platform needs to be considered
- At some stage scalability and availability will become an issue.

Typical Event Observer Model – Using Dumb Pipe



Implementing Dumb Pipes

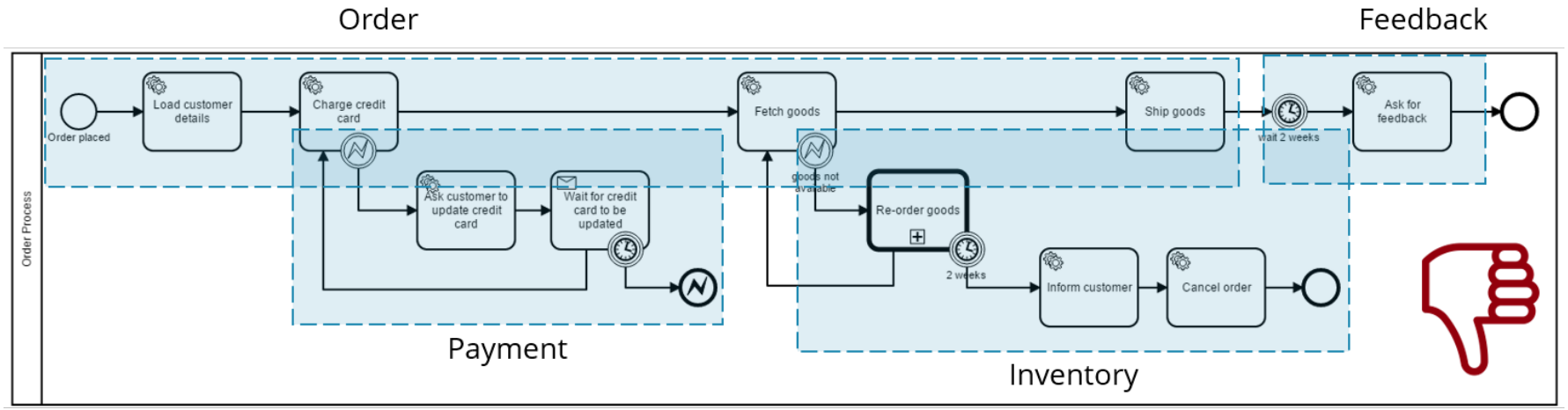
The standard way for distributing this type of workload is to pass event messages using a broker service, ideally one that implements a queue,
e.g. RabbitMQ, ZeroMQ, Kafka, or even Redis Pub/Sub

Amazon provides Amazon Simple Notification Service (SNS), and Amazon Simple Queue Service (SQS) as a fully managed solution for broadcast communication.

These services allow a producing service to make one request to an SNS topic to broadcast an event, while multiple SQS queues can be subscribed to that topic, with each queue connected to a single microservice that consumes and responds to that event.

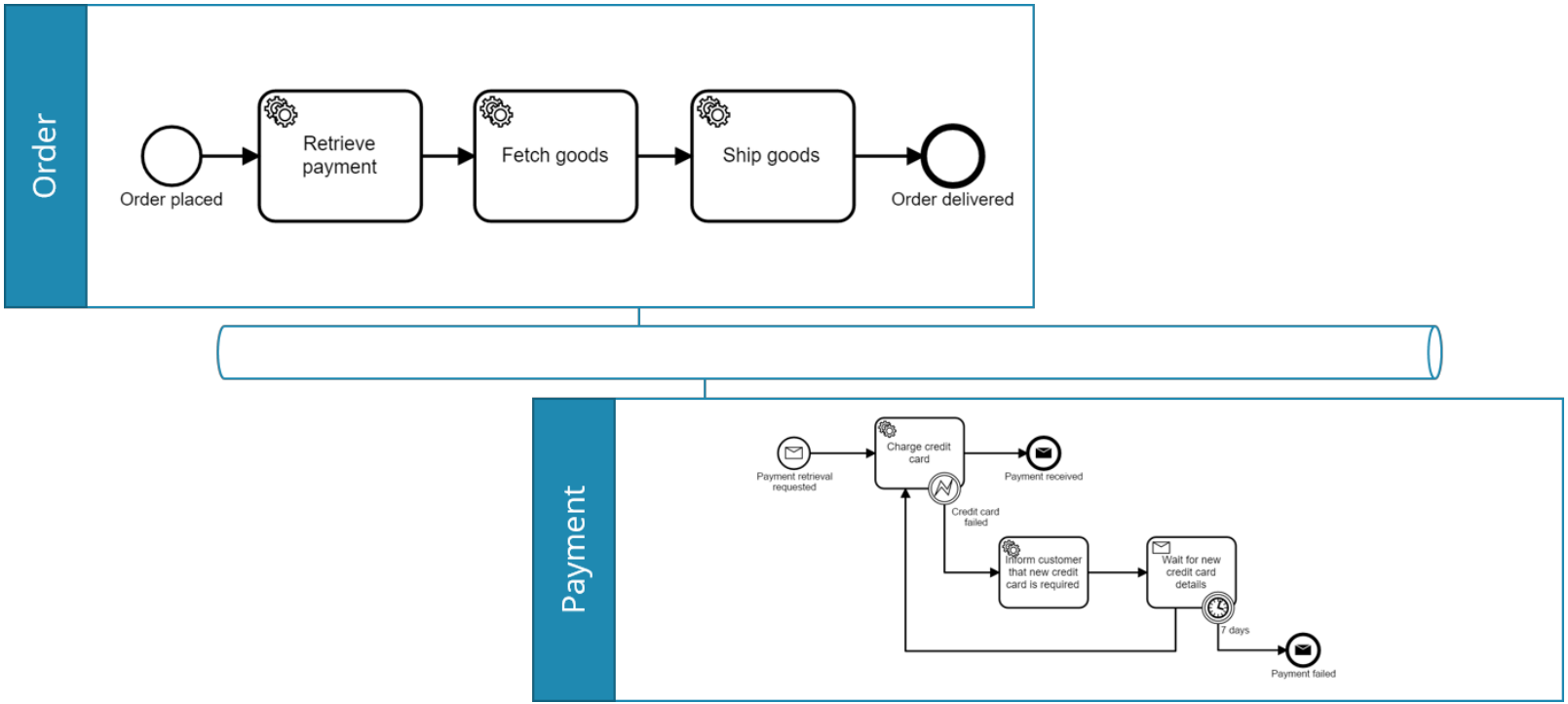
Problem with Monolithic BPM

- Processes cross Bounded Contexts
- As a result no one wants to own the end to end process.
- Hard to get process to work across autonomous teams.
- Shared infrastructure result in dependencies between teams.



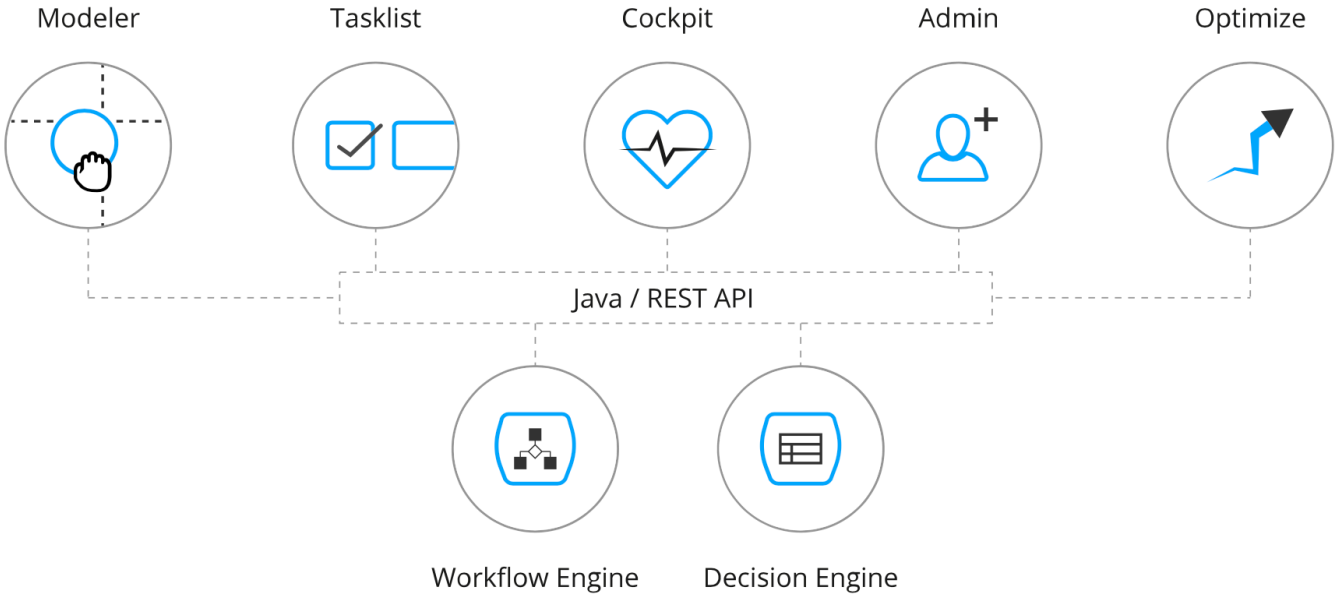
Solution move process into its Bounded Context.

- Get each service to own its process
- Run multiple process engines in your environment as every microservice needs its own

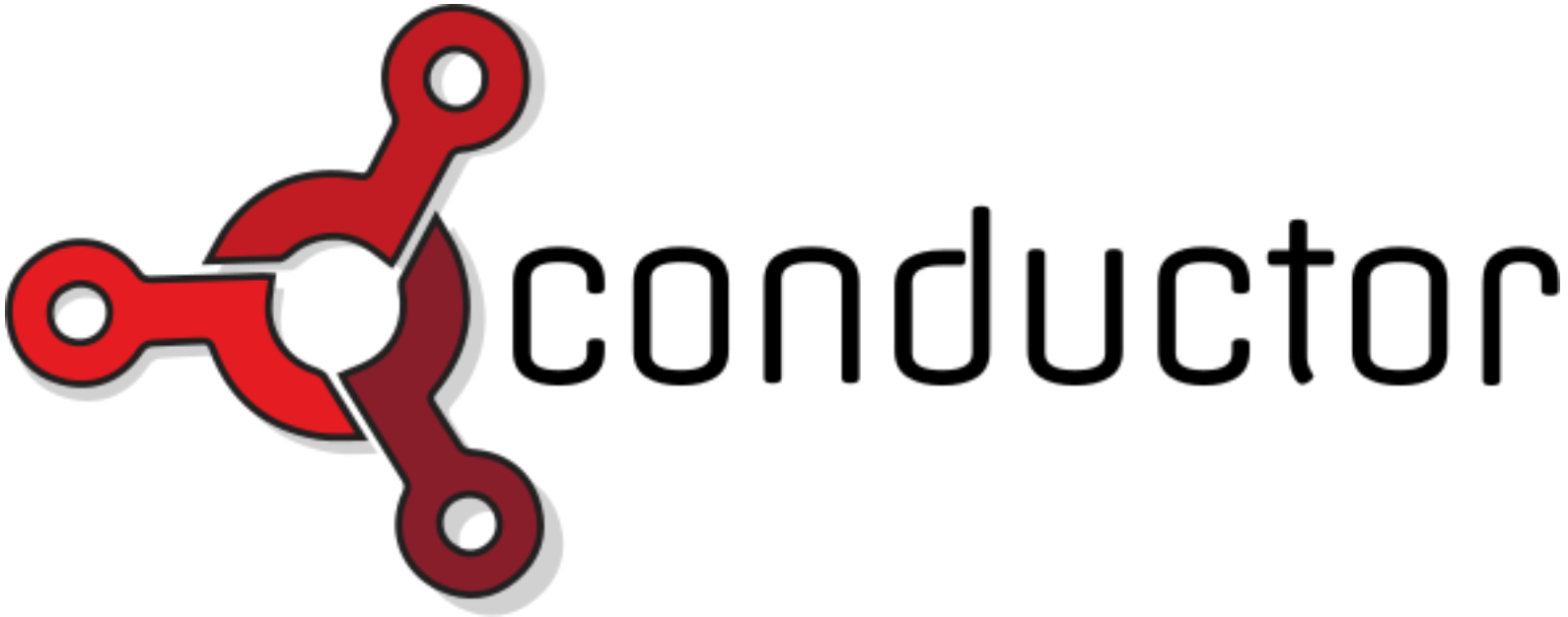


Camuda

- Camuda is a lightweight, open source process and decision engine.



Netflix Conductor



Orchestration Vs Choreography (observer)

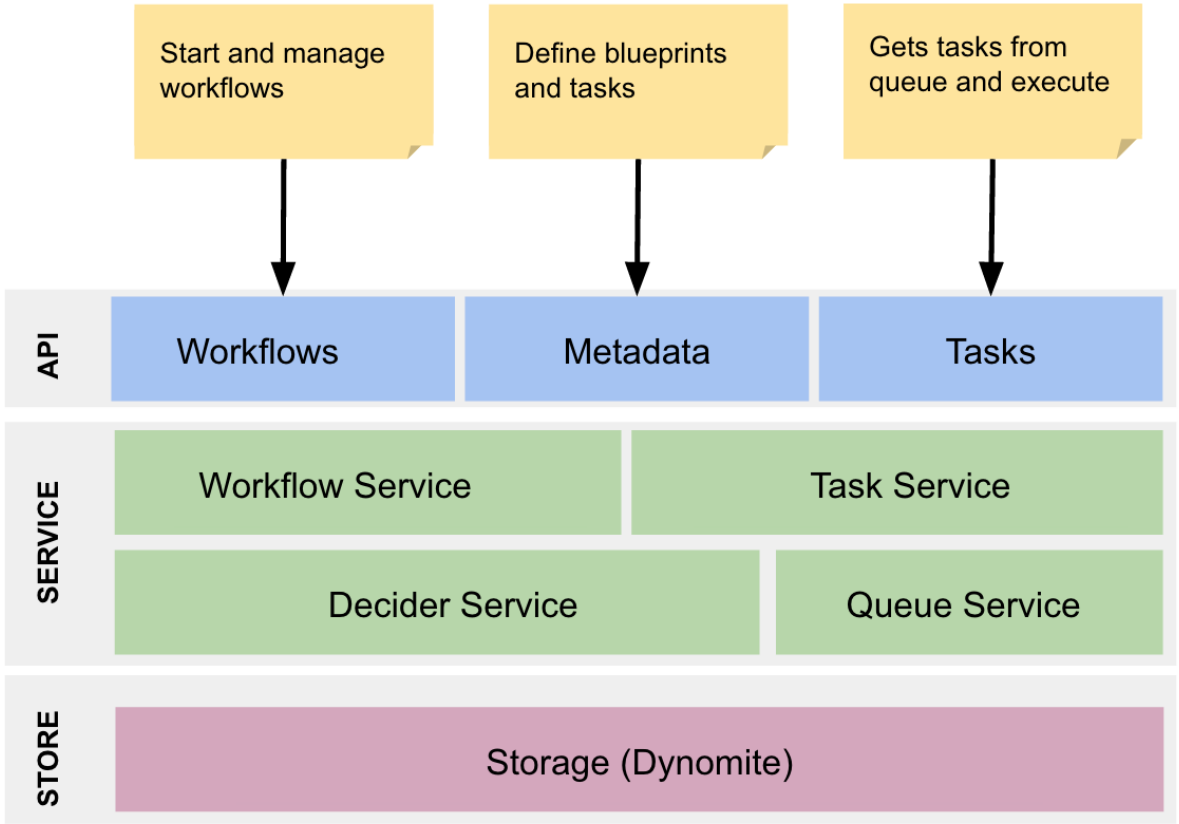
Often suggested that Choreography is less coupled than Orchestration approaches. Often Pub/Sub Events, via dumb pipes, are used to signal a next step by an interested party that is observing the events. There is no central organisation of 'process', just a sequence based on response to events.

However, Netflix found that choreography was harder to scale with growing business needs and complexities.

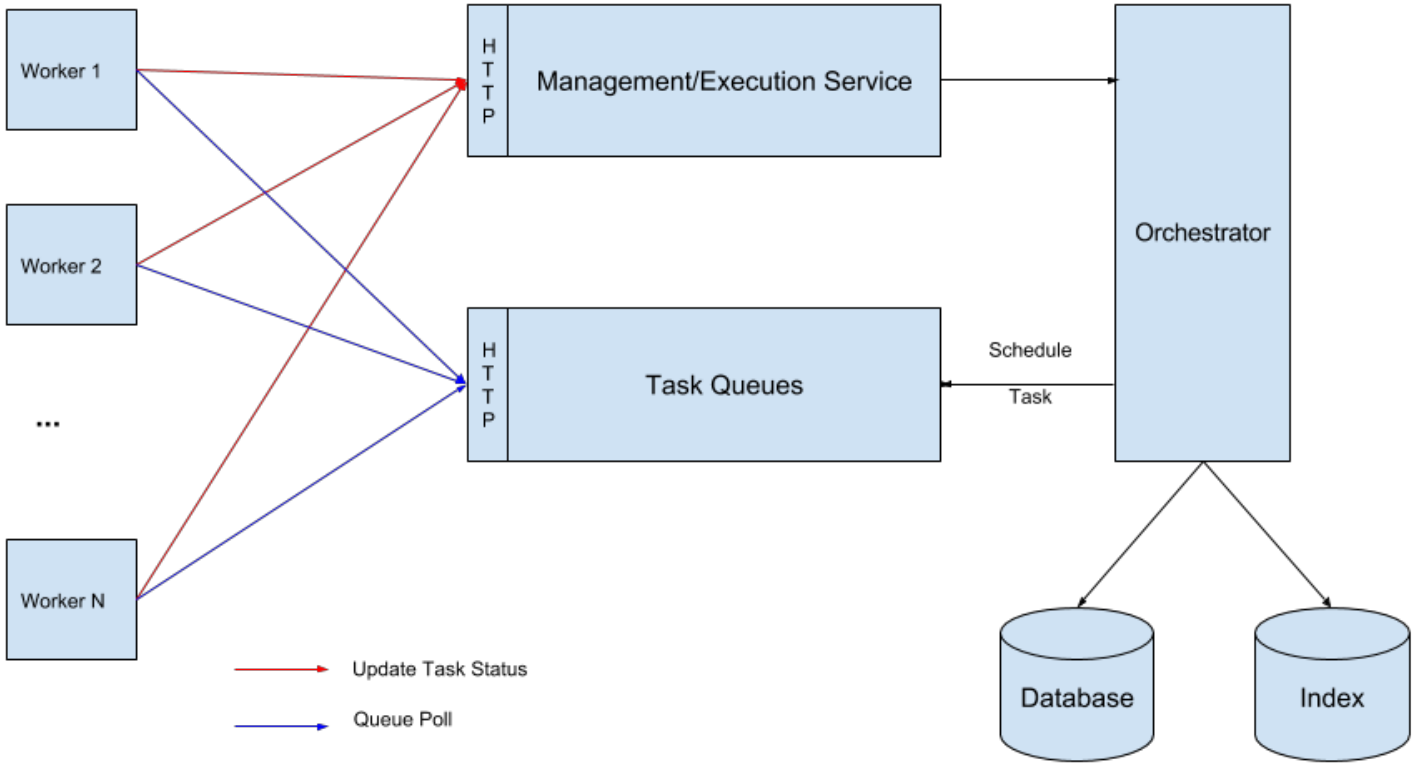
Pub/sub model worked for simplest of the flows, but quickly highlighted some of the issues associated with the approach:

- Process flows are "embedded" within the code of multiple applications
- Often, there is tight coupling and assumptions around input/output, SLAs etc, making it harder to adapt to changing needs
- Almost no way to systematically answer "What is remaining for a movie's setup to be complete"?

Conductor Architecture



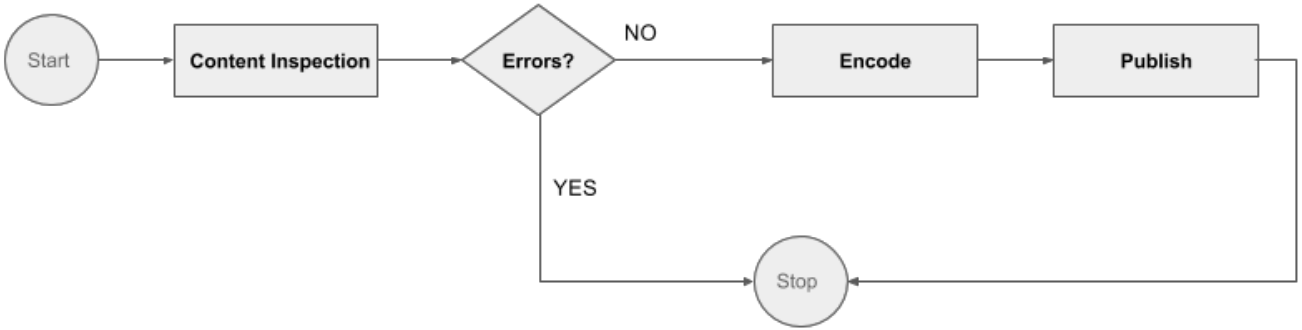
Worker Communication



A Workflow Definition

```
{
        "name": "workflow_name",
        "description": "Description of workflow",
        "version": 1,
        "tasks": [
                {
                "name": "name_of_task",
                "taskReferenceName": "ref_name_unique_within_blueprint",
                "inputParameters": {
                "movieId": "${workflow.input.movieId}",
                "url": "${workflow.input.fileLocation}"
                },
                "type": "SIMPLE",
                ... (any other task specific parameters)
        },
        {}
        ...
        ],
        "outputParameters": {
        "encoded_url": "${encode.output.location}"
        }
}
```

Example: encode and deploy workflow



There are a total of 3 worker tasks and a control task (Errors) involved:

1. Content Inspection: Checks the file at input location for correctness/completeness
2. Encode: Generates a video encode
3. Publish: Publishes to CDN

- The tasks are implemented by different workers which are polling for pending tasks using the task APIs. These are ideally idempotent tasks that operate on the input given to the task, performs work, and updates the status back.
- As each task is completed, the Decider evaluates the state of the workflow instance against the blueprint and identifies the next set of tasks to be scheduled, or completes the workflow if all tasks are done.

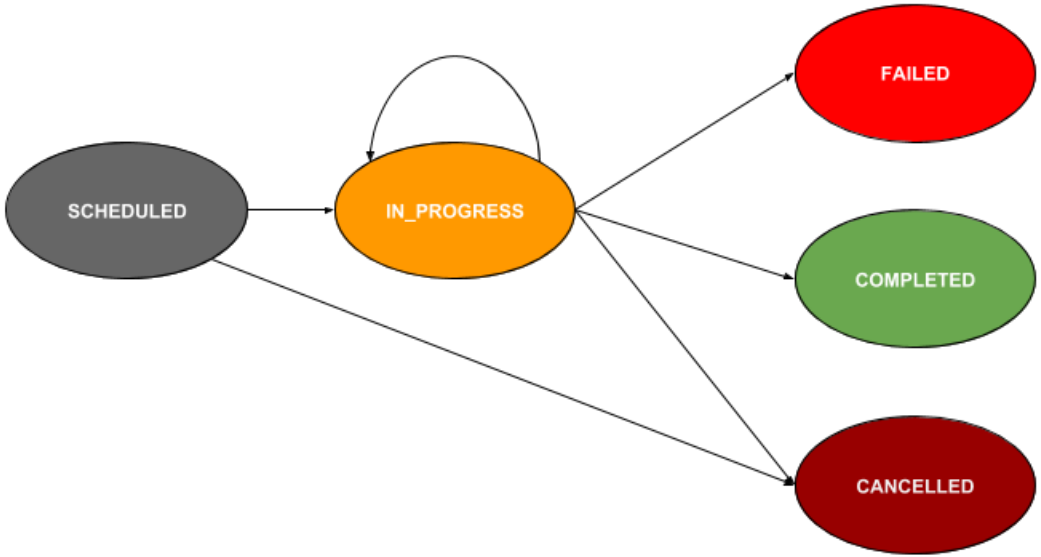
System Tasks

Name	Purpose
DYNAMIC	A worker task which is derived based on the input expression to the task, rather than being statically defined as part of the blueprint
DECIDE	Decision tasks - implements case...switch style fork
FORK	Forks a parallel set of tasks. Each set is scheduled to be executed in parallel
FORK_JOIN_DYNAMIC	Similar to FORK, but rather than the set of tasks defined in the blueprint for parallel execution, FORK_JOIN_DYNAMIC spawns the parallel tasks based on the input expression to this task
JOIN	Complements FORK and FORK_JOIN_DYNAMIC. Used to merge one of more parallel branches*
SUB_WORKFLOW	Nest another workflow as a sub workflow task. Upon execution it instantiates the sub workflow and awaits it completion
EVENT	Produces an event in a supported eventing system (e.g. Conductor, SQS)

Worker Tasks

Worker tasks are implemented by application(s) and run in a separate environment from Conductor. The worker tasks can be implemented in any language. These tasks talk to Conductor server via REST API endpoints to poll for tasks and update its status after execution.

Worker Lifecycle:



DaaS Lambda, Pipes and Filters Style

Data as a Services (DaaS)

- DaaS "*is a cousin of software as a service (SaaS). Like all members of the "as a service" (aaS) family, DaaS builds on the concept that the product (data in this case) can be provided on demand to the user regardless of geographic or organizational separation of provider and consumer. Additionally, the emergence of service-oriented architecture (SOA) and the widespread use of application programming interface (API) has also rendered the actual platform on which the data resides irrelevant.*" – Wikipaedia

- Microservices and API approaches can be applied to DaaS. Note however, that Data only Services are not recommended for Business Application Microservice Architectures as they result in 'Anemic Services' or the Anemic Data Model Anti-pattern, where an entity has data but no business functions or behaviour. DaaS are a special case.

- Traditional Batch ETL and Data Warehouse / Analytics approaches are not working; the data is usually old, not up to date, and integration is very complex; data is now arriving too fast.

- Today the volumes of data cannot be managed by these 'old' approaches. New Approaches to Streaming Data over Batch approaches are required.

Lambda Architecture

“is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods. This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs may be joined before presentation. The rise of lambda architecture is correlated with the growth of big data, real-time analytics, and the drive to mitigate the latencies of map-reduce.” - Wikipaedia

Data Streaming Definition

- Streaming Data is data that is available when a client needs it, not necessarily in real time. It is data that needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows.
  - The data is used for finding patterns, correlation, analytics, filtering, sampling, machine learning, data mining etc.
- The data could be generated continuously by thousands of data sources and includes, log files, business events, IOT devices, stock market data, ecommerce purchases and so on.

Data Microservices Approach

- The approach is to break the problem down into small services that each perform a single task or capability, e.g. data collection, correlation, filtering, storage.
- The tasks run as microservices and so are distributed, scalable.
- The task are linked together in orchestrated pipelines.
- Map reduce is used with Key-Value pairs.
- Streams are used to link the processes, for example using Kafka.

Adding Processor Node

- In the prior example we discussed a Service listening to a topic for events from a source

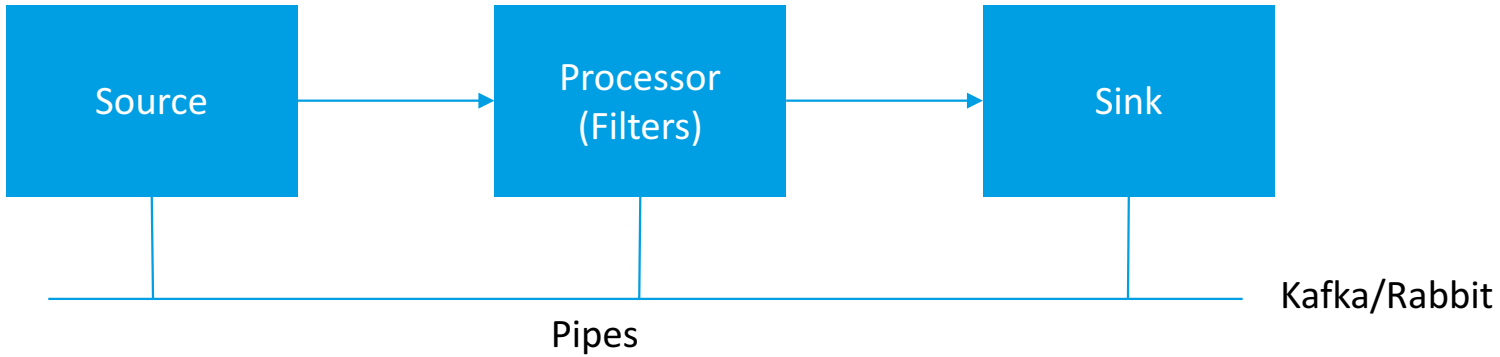
Kafka Broadcast:

- Only one consumer in a consumer group can pull the message. But **all** consumer groups get the messages.
- If you want all your consumers to get the messages, assign them different consumer groups. Each message goes to every consumer group, but within a group, it goes to only one consumer.

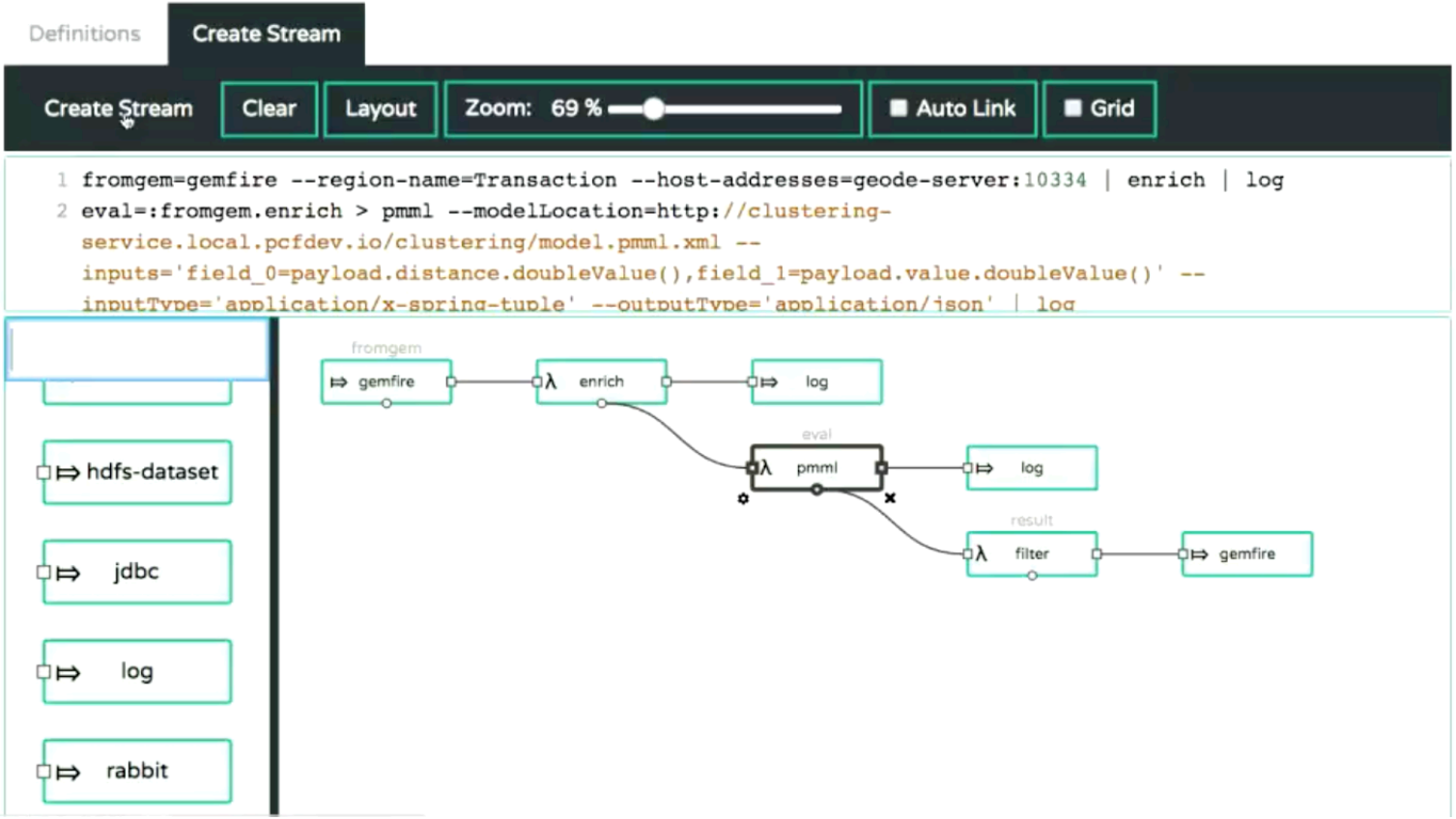
Pipes and Filters Microservices

Spring Cloud Stream provides three predefined interfaces out of the box(either an input channel, an output channel, or both).

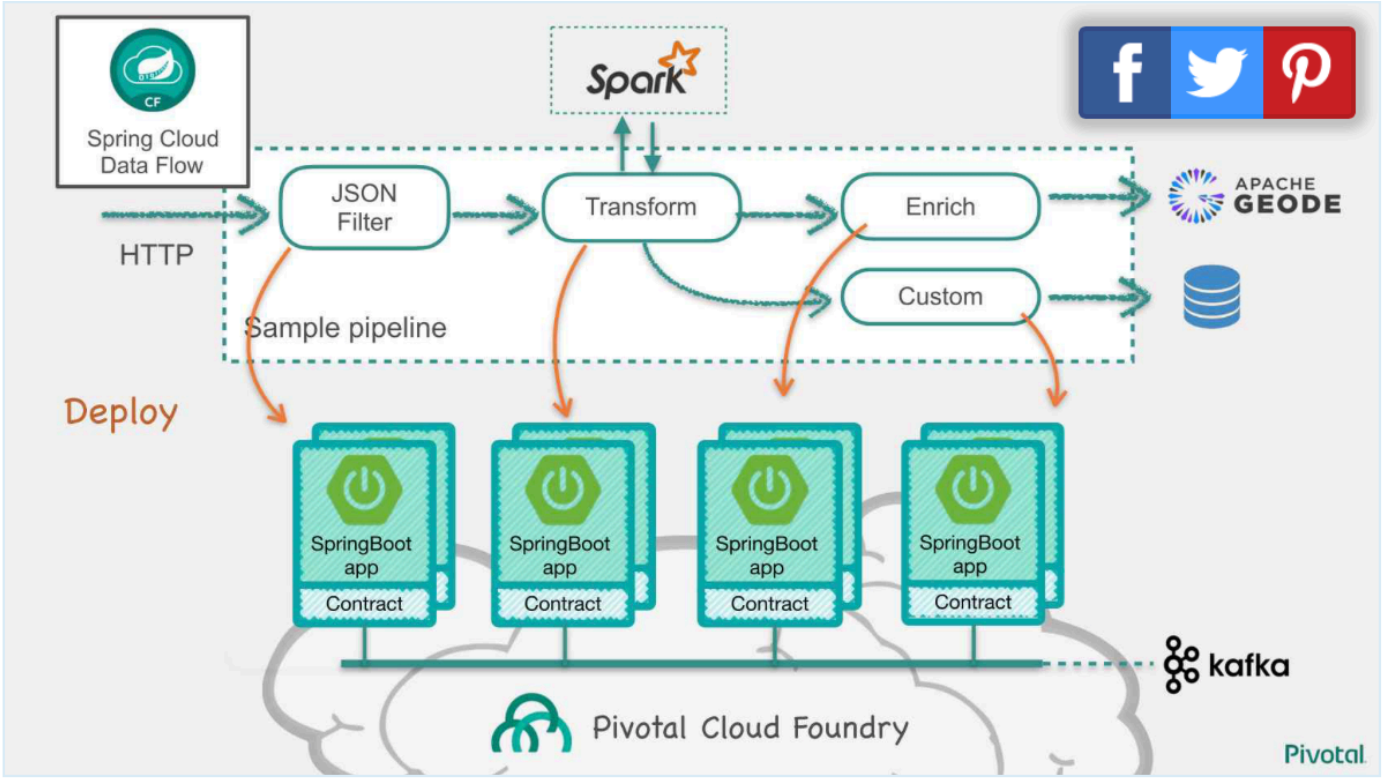
- Code is message bus agnostic
- Orchestrated with Spring Cloud DataFlow



Spring Cloud Data Flow



Spring Cloud Data Flow - Orchestrate



Data Service Architecture