Event-driven systems

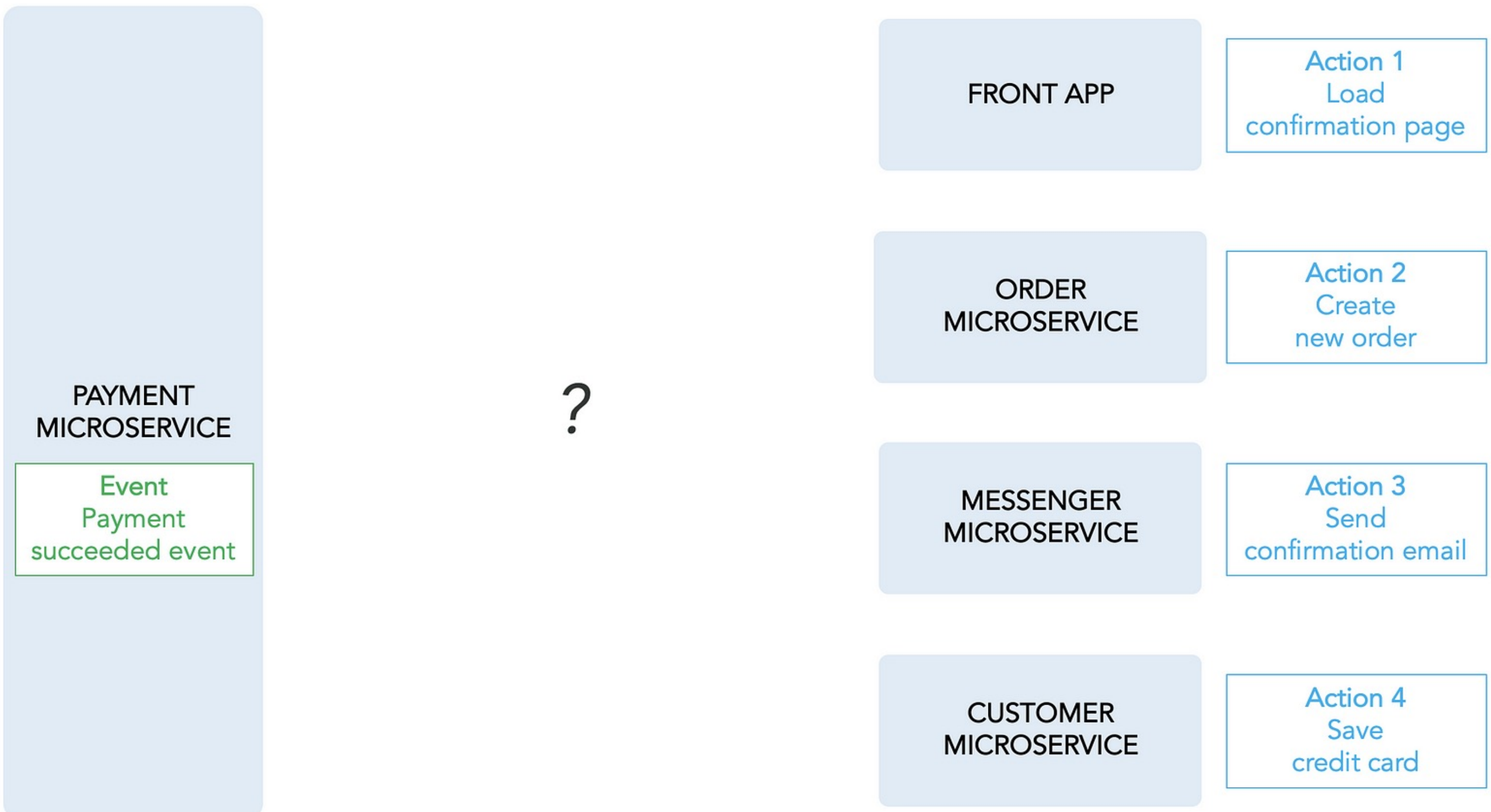
UA.DETI.IES

Scaling with Microservices

- **Application development** is currently based on concepts like **microservices** and **serverless** functions
  - We now live in a **cloud-native world**, and these models are a natural fit for distributed cloud-based environments
  - But simply building and deploying services and functions is not enough

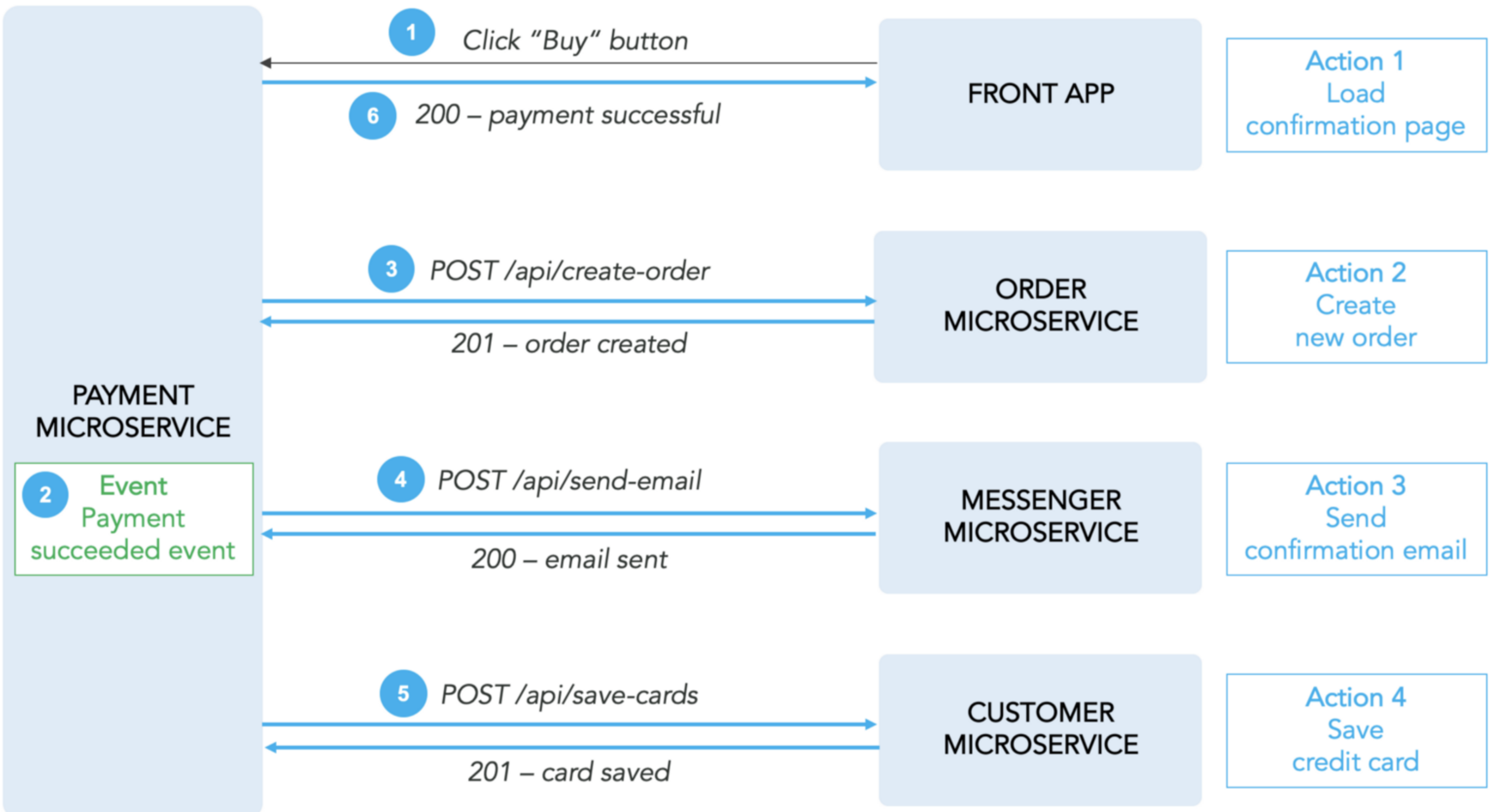
- On its own, a single microservice does not accomplish much
  - **We also need a way to wire up those components**
    - i.e., to connect them so they can exchange data, forming a true application
  - In SE, this is one of the most important architectural decisions to make

How to notify every Microservice?

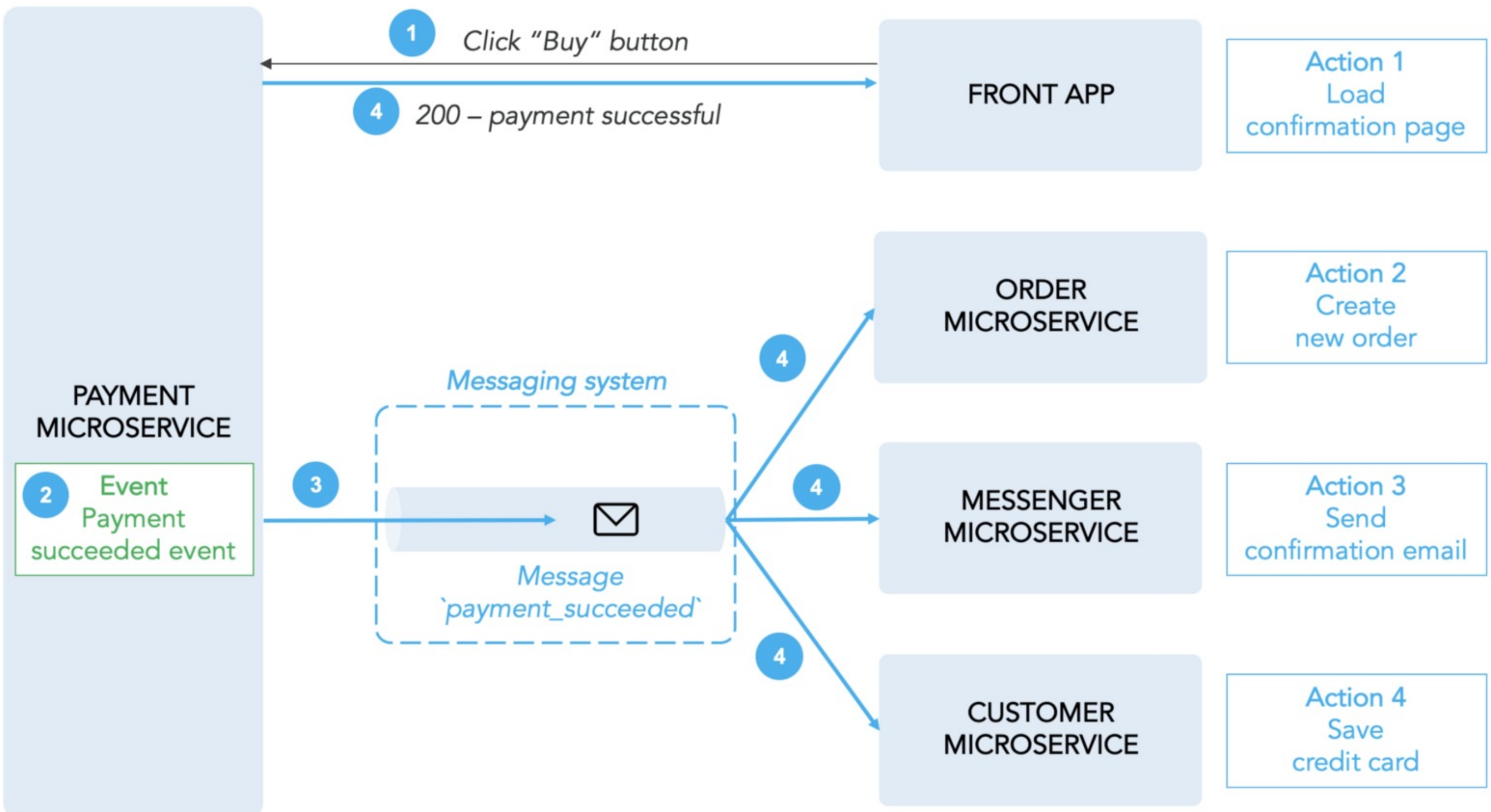


https://blog.theodo.com/2019/08/event-driven-architectures-rabbitmq/

Request-driven architecture



Event-driven architecture



Event-driven architecture

- For controlling these services, various mechanisms were developed over the years, such as **message queues** and **enterprise service buses** (ESBs).
  - E.g. **RabbitMQ**, **WSO2**.
- More recent offerings, the concept of **streaming data** have also emerged.
  - E.g. Apache **Kafka**
- This latter category is growing
  - Because streaming data is seen as a useful tool for implementing **event-driven architecture**
  - <u>a software design pattern in which application data is modeled as streams of events, rather than as operations on static records.</u>

What is an event?

- **Events** are things that happen, within a software system or, more broadly, during the operation of a business or other human process.
  - e.g., a sensor reports a temperature change, a user clicks their mouse, a customer deposits a check into a bank account.
- The concept of events in software systems closely aligns with how most of us think about our day-to-day lives.
  - Organizing around events makes it easier to develop business logic that accurately models real-world processes.
  - It helps reducing the number of one-to-one connections within a distributed system increasing the value of the microservices.
- An **event-driven architecture** allows generating, storing, accessing and reacting to these events.

Events …

Events vs. queries and commands

- **Queries** are a request to look something up
  - Unlike events or commands, queries are free of side effects; they leave the state of the system unchanged.
- **Commands** are actions
  - Requests for some operation to be performed and will change the state of the system.
  - Synchronous and typically indicate completion.

	Behavior/state change	**Includes a response**
Command	Requested to happen	Maybe
Event	Just happened	Never
Query	None	Always

Event-driven patterns – notification

- A service sends **events to notify** other systems of a change in its domain
  - For example, a user account service might send a notification event when a new login is created
- What other systems choose to do with that information is largely up to them
  - The service that issued the notification just carries on with its business
- Notification events **usually do not carry much data**
  - Resulting in a loosely coupled system with minimal network traffic spent on messaging

Event-driven patterns – state transfer

- A step up from simple notification, in this model the recipient of an **event also receives the data** it needs to perform further work
  - E.g., the user account service might issue an event that includes a data packet containing the new user's login ID, full name, hashed password, and other pertinent details.

- This model can be appealing to developers familiar with RESTful interfaces.
  - But, depending on the complexity of the system, it can lead to a lot of data traffic on the network and data duplication in storage

Event-driven patterns – sourcing

- The goal of **event-sourcing** is to represent every change of state in a system as an event, each recorded in chronological order

- In so doing, the **event stream itself becomes the principal source of truth for the system**
  - E.g., it should be possible to "replay" a sequence of events to recreate the state of a SQL database at a given time

- This model presents a lot of possibilities, but it can be challenging to get right
  - particularly when events require participation from external systems

Event-driven advantages (over REST)

- **Asynchronous**
  - Allows resources to move to the next task once a unit of work is complete
  - Events are queued or buffered which prevents consumers from putting back pressure on producers or blocking them

- **Loose Coupling**
  - Services operate independently, without knowledge of other services, including their implementation details and transport protocol
  - Services under an event model can be updated, tested, and deployed independently and more easily

https://dzone.com/articles/best-practices-for-event-driven-microservice-archi

Event-driven advantages (over REST)

- **Easy Scaling**
  - Since the services are decoupled and typically perform only one task, tracking bottlenecks and scaling a service is easier.

- **Recovery Support**
  - Can recover lost work by “replaying” events from the past.

https://dzone.com/articles/best-practices-for-event-driven-microservice-archi

Event-driven disadvantages

- They are easy to **over-engineer** by separating concerns that might be simpler when closely coupled
  - they can require significant upfront investment, and often result in additional complexity, service contracts or schemas, polyglot build systems, and dependency graphs

- **Complex data and transaction** management
  - Typically, do not support ACID transactions
  - Systems must carefully handle inconsistent data between services, incompatible versions, duplicate events

- Even with these drawbacks, …
  - An event-driven architecture is usually the better choice for enterprise-level microservice systems
  - The pros—scalable, loosely coupled, dev-ops friendly design—outweigh the cons

Event-driven Anti-Patterns

- Depending on Guaranteed Order and Delivery
  - Events are asynchronous
    - Including assumptions of order or duplicates will add complexity and will negate the key benefits of the event-based architecture.

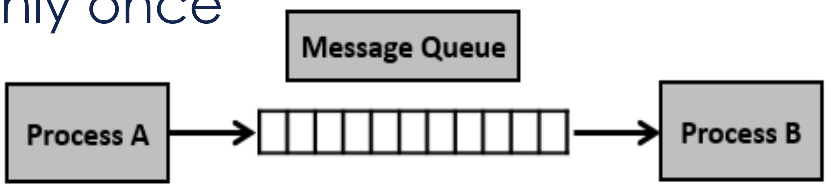
- Premature Optimization
  - Most products start off small and grow over time
    - Consider a simple architecture but include the necessary separation of concerns so that you can swap it out as your needs grow.

- Expecting Event-Driven to Fix Everything
  - Event-driven architecture to fix all the problems
    - It can't fix core problems such as a lack of automated testing, poor team communication, or outdated dev-ops practices.

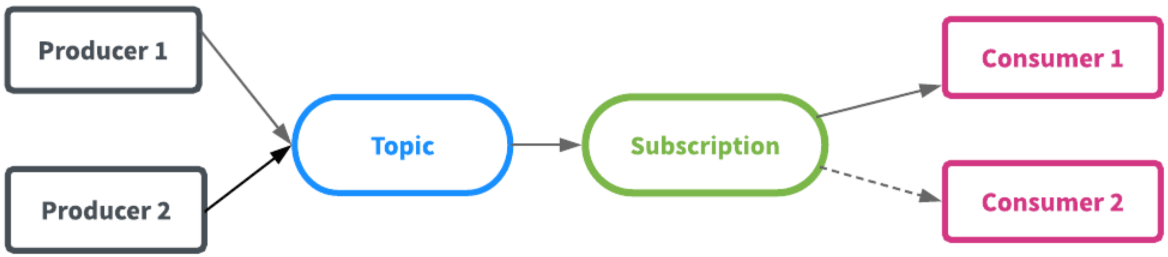
Messaging systems

Messaging systems – models

- **Point to Point** – Message queue
  - Messages are sent to a queue to pre-defined receivers
  - One-to-one relationship between sender and consumer
  - Each message is consumed only once

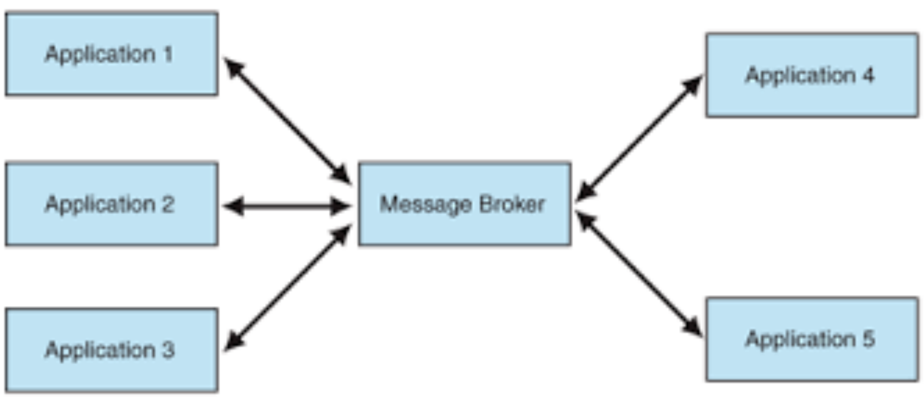


- **Publish-Subscribe**
  - Each message is published to a topic, and every application that subscribes to that topic gets a copy of all messages published to it.
  - Message producers are also known as publishers and consumers are known as subscribers.



Messaging systems

- Managing the messages' flow
  - Messages are "put into" a source queue
  - They are then "taken from" a destination queue
  - How/Who/What moves a message from a source queue to a destination queue?
- Queue Manager / **Message Broker**
  - Function as message-queuing "relay" that interact with distributed applications



Message broker

- Message Broker is built to extend MQ
  - it can understand the content of each message that it moves through the Broker.

- Message Broker can do the following:
  - divide the publisher and consumer
  - store and route the messages between services
  - converts between different transport protocols
  - identifies and distributes business events from disparate sources

Message broker

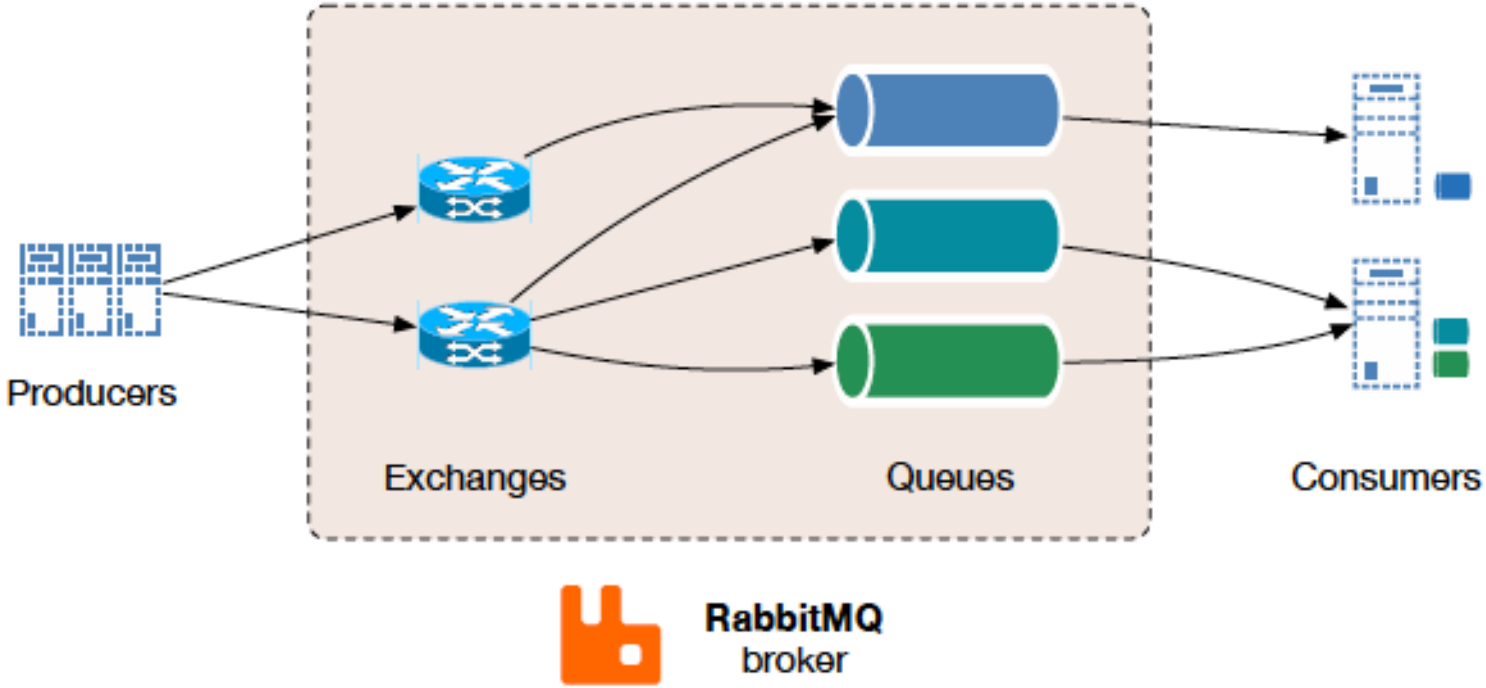
- When is a message broker needed?
  - If we want to **control data feeds**, e.g., the number of registrations in a system
  - When the task is to **put data to several applications** and avoid direct usage of their API
  - When there is a need to **complete processes in a defined order** like a transactional system

- There are many messaging tools…
  - E.g. Apache ActiveMQ, RabbitMQ

- ... and protocols
  - E.g. AMQP (Advanced Message Queuing Protocol), MQTT (MQ Telemetry Transport)

RabbitMQ

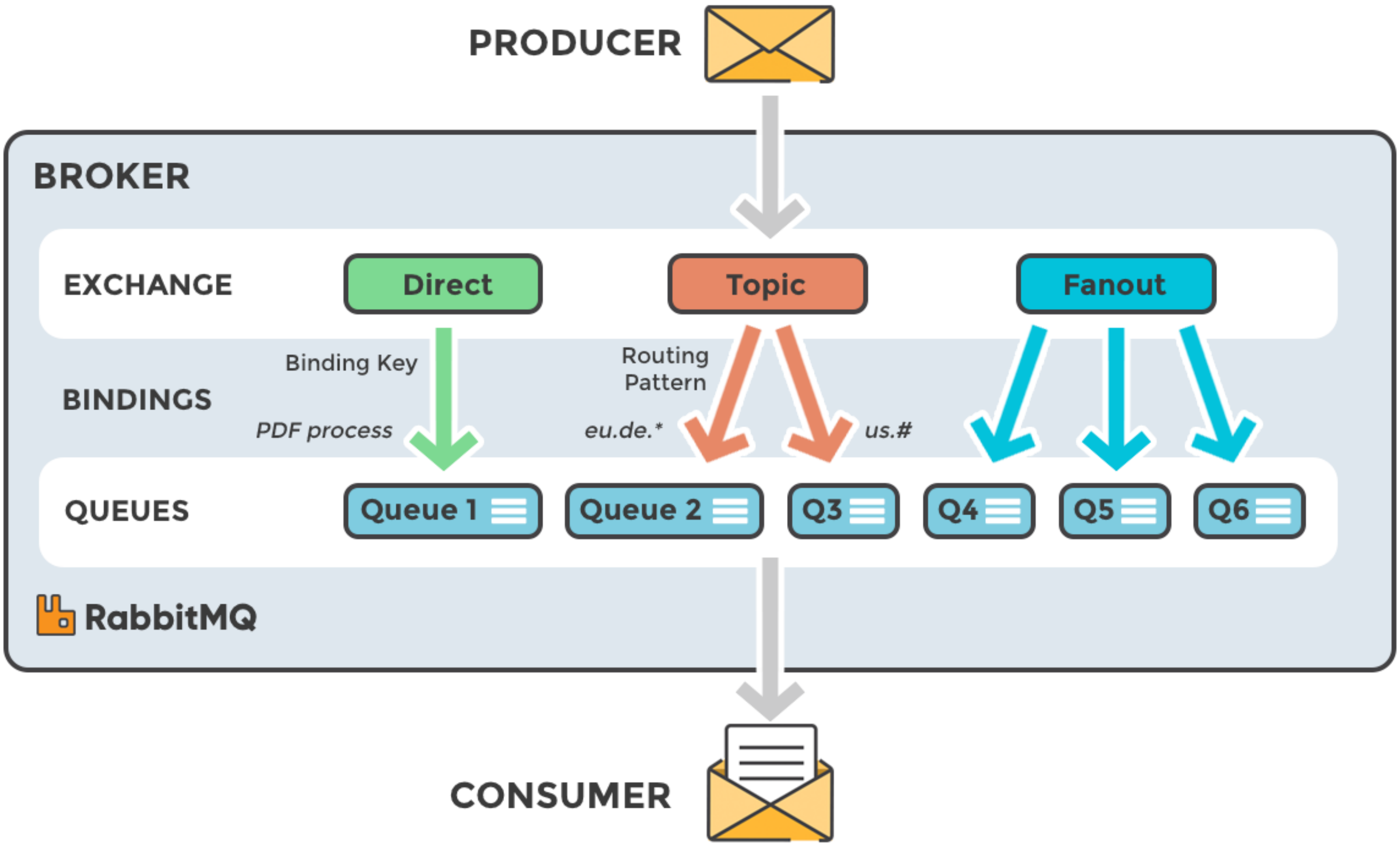
- RabbitMQ is one such open-source message broker software that implements AMQP.



RabbitMQ – Main concepts

- **Messages** contains attributes (like headers in a request) and a payload (the message content).
- Messages are published to an entity, **exchange**, which distribute the messages to **queues** (or Topics).
- The rules for delivering the messages to the right queues are defined through
  - **bindings** (links between exchanges and queues), and
  - **routing keys** (a specific message attribute used for routing).
- Messages stored in queues are
  - delivered continually to subscribers, or
  - fetched by consumers on demand.

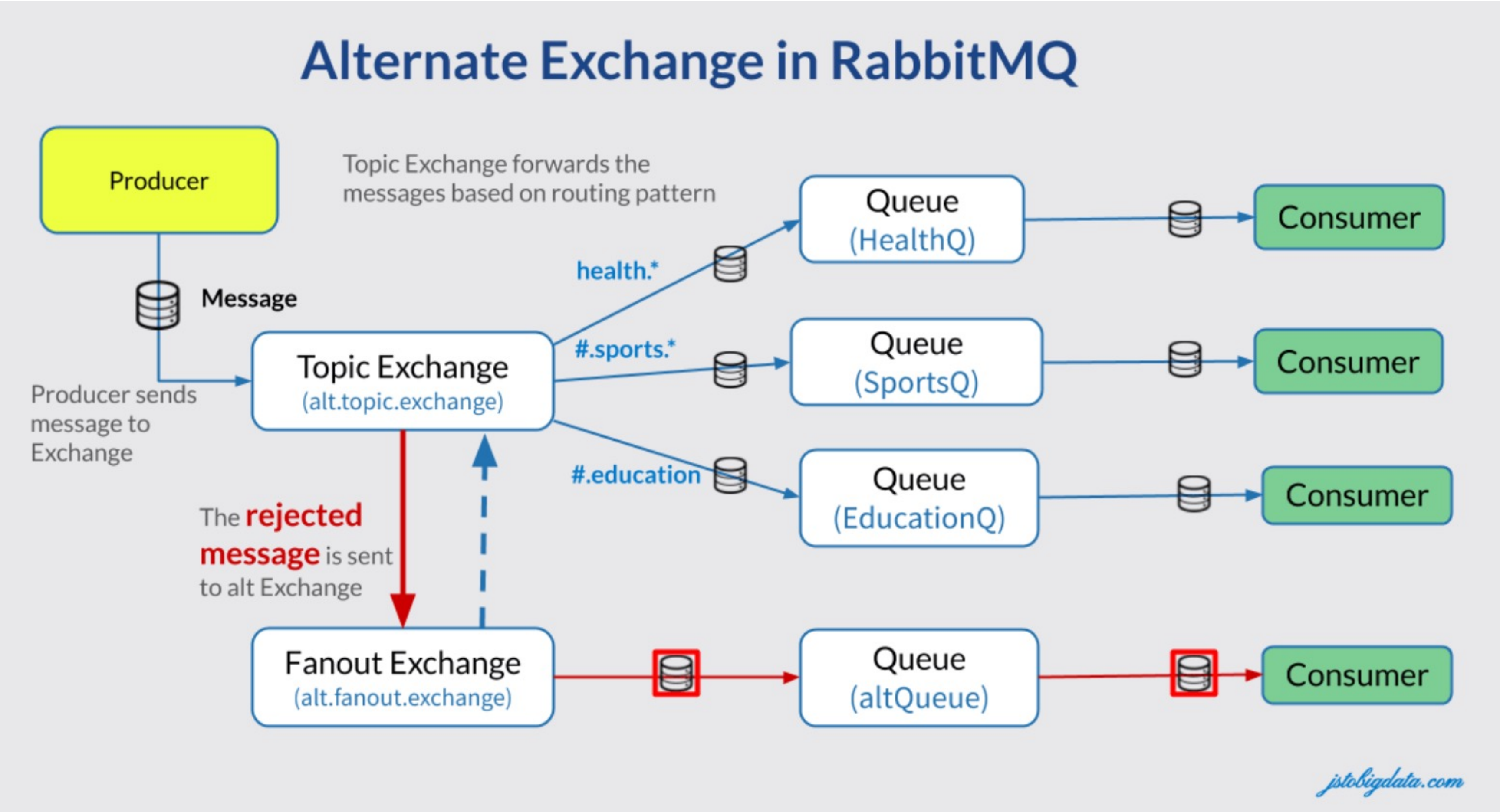
RabbitMQ – Main concepts



RabbitMQ – Exchange types (AMQP)

- **Direct Exchange** - It routes messages to a queue by matching **routing key** equal to **binding key**.
- **Topic Exchange** – It routes messages to multiple queues by a partial matching of a routing key. It uses patterns to match the routing and binding key.
- **Fanout Exchange** - It ignores the routing key and sends message to all the available queues.
- **Headers Exchange** – It uses message header instead of routing key.
- **Default (Nameless) Exchange** - It routes the message to queue name that exactly matches with the routing key.

RabbitMQ – Exchange types



Spring Boot – Messaging with RabbitMQ

- Set up the RabbitMQ Broker
- Spring Initializr
- E.g. RabbitMQ dependency

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-amqp -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-amqp</artifactId>
<version>2.3.2.RELEASE</version>
</dependency>
```

- https://spring.io/guides/gs/messaging-rabbitmq/

Example excerpt …

```java
@SpringBootApplication
public class MessagingRabbitmqApplication {
  static final String topicExchangeName = "spring-boot-exchange";
  static final String queueName = "spring-boot";

  @Bean
  Queue queue() {
    return new Queue(queueName, false);
  }

  @Bean
  TopicExchange exchange() {
    return new TopicExchange(topicExchangeName);
  }

  @Bean
  Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("foo.bar.#");
  }
```

UNIVERSIDADE DE AVEIRO

Hello world – Sender

```
public class Sender {

    @Autowired
    private RabbitTemplate template;

    @Autowired
    private Queue queue;

    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
        String message = "Hello World!";
        this.template.convertAndSend(queue.getName(), message);
        System.out.println(" [x] Sent '" + message + "'");
    }
}
```

Hello world – Receiver

```
@RabbitListener(queues = "hello")
public class Receiver {

    @RabbitHandler
    public void receive(String in) {
        System.out.println(" [x] Received '" + in + "'");
    }
}
```

- Tutorial:
  - https://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html

Event streaming systems
Apache Kafka

Stored records *vs* Messaging

ETL/Data Integration

–	+
Batch	**High Throughput**
Expensive	**Durable**
Time Consuming	**Persistent**
	Maintains Order

Messaging

+	–
Fast (Low Latency)	Difficult to Scale
	No Persistence
	Data Loss
	No Replay

https://pt.slideshare.net/ConfluentInc/what-is-apache-kafka-and-what-is-an-event-streaming-platform

The Event Streaming Paradigm

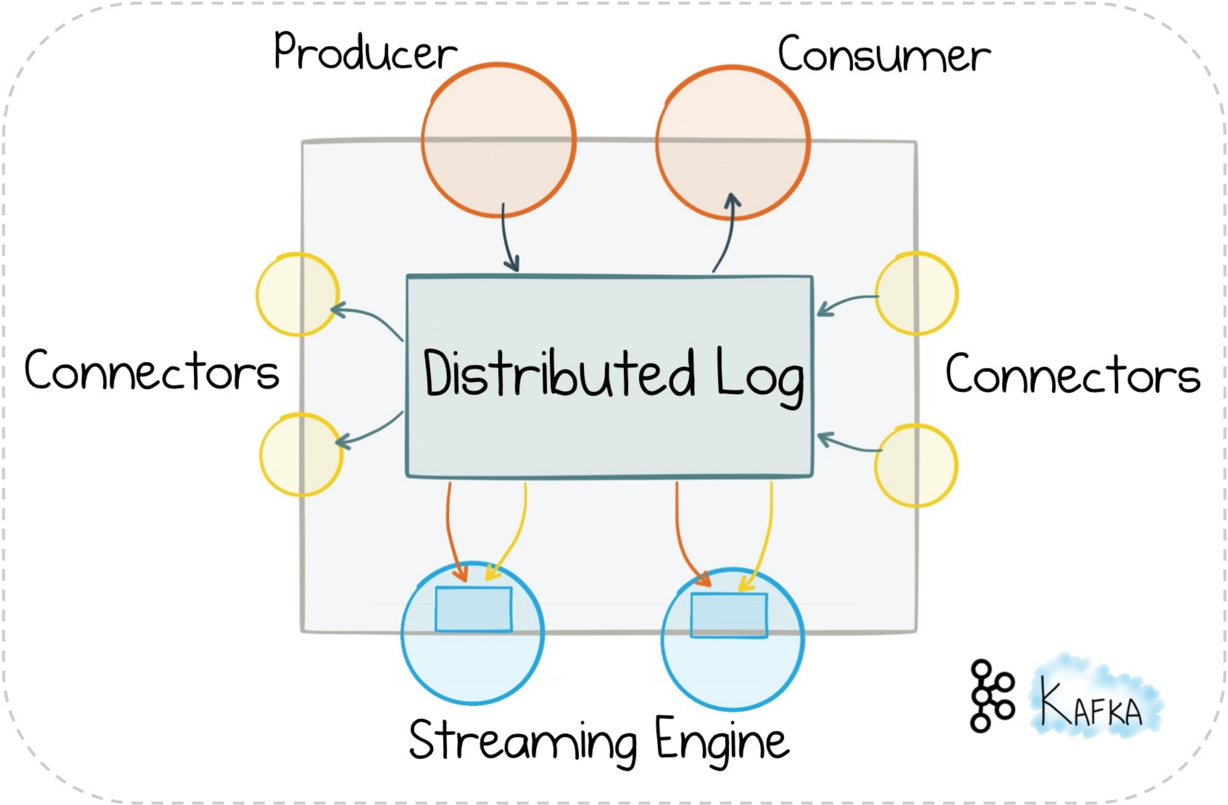
The Event Streaming Paradigm

To rethink data as not stored records or transient messages, but instead as a continually updating stream of events

https://pt.slideshare.net/ConfluentInc/what-is-apache-kafka-and-what-is-an-event-streaming-platform

Apache Kafka

- Apache Kafka is made of distributed, immutable, append-only commit logs

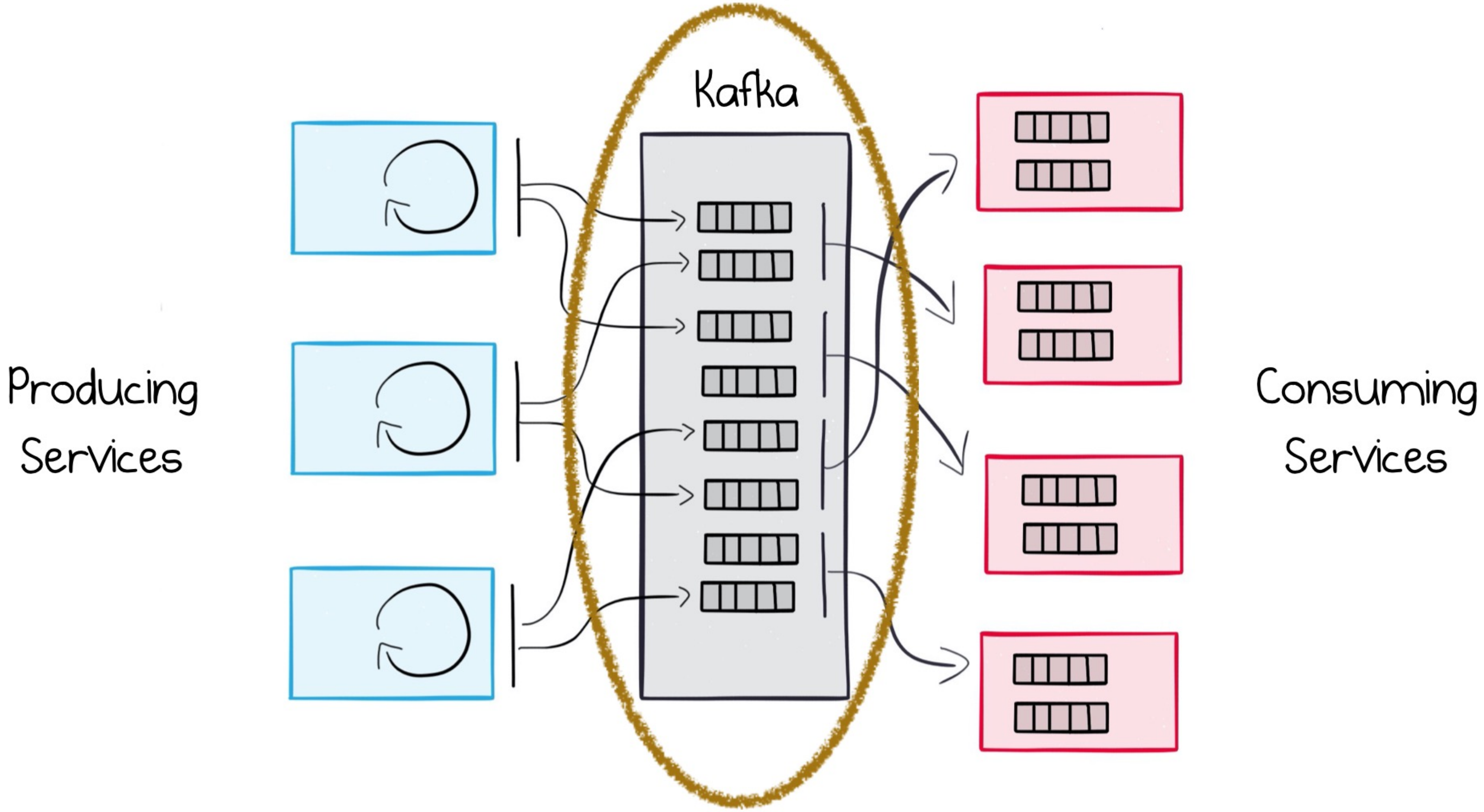


The Distributed Log

Shard on the way in

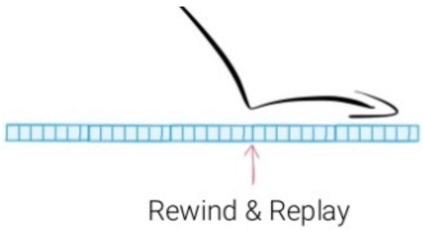
Each shard is a queue

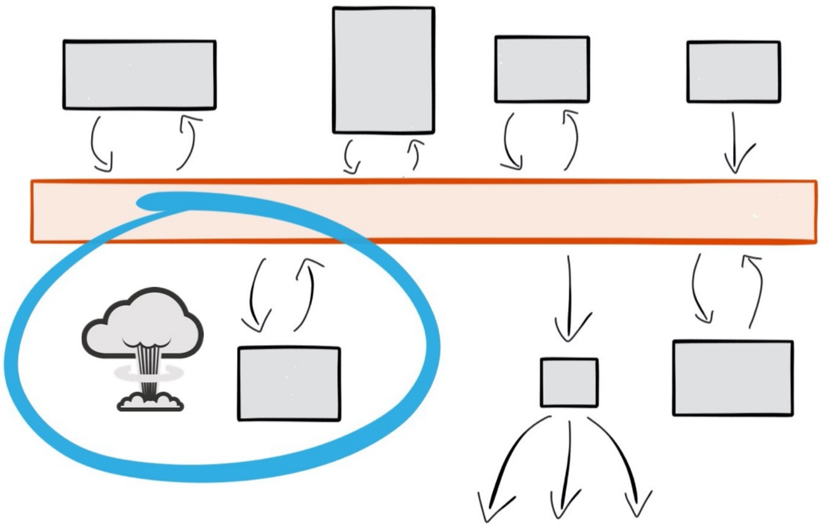
Consumers share load



The Distributed Log

- The Service Backbone
  - Scalable/Load Balanced, Fault Tolerant, Concurrent, Strongly Ordered, Stateful





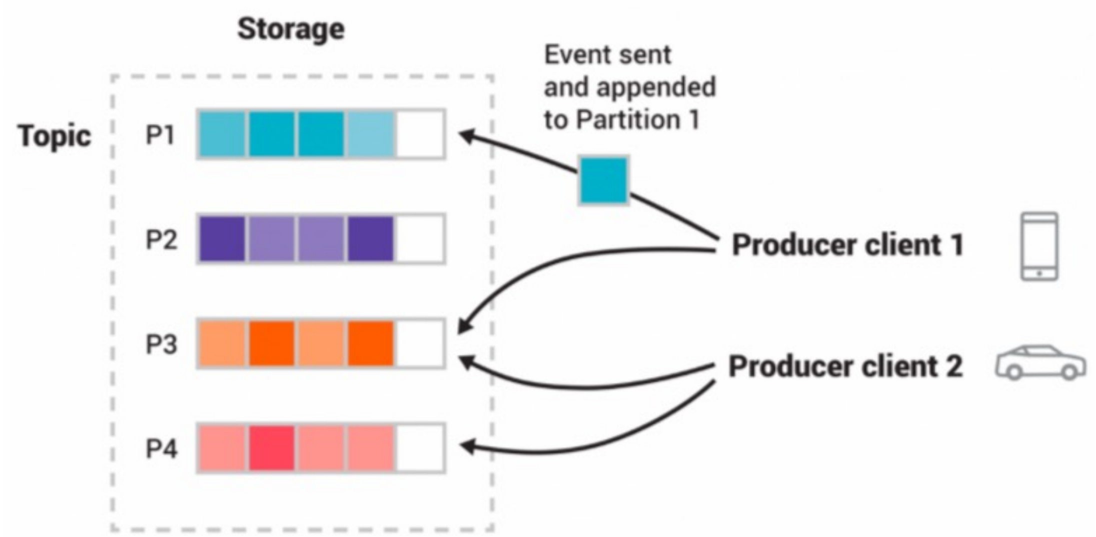
- A place to keep data-on-the-outside
  - When sending data across services, it is outside the normal "trust" boundary.

Events and topics

- Each **event** has a key, value, timestamp, and optional metadata headers.

```
Event key: "Alice"
Event value: "Made a payment of $200 to Bob"
Event timestamp: "Jun. 25, 2020 at 2:06 p.m."
```

- Events are organized and durably stored in **topics**
- Topics are **partitioned**
  - a topic is spread over "buckets" located on different Kafka brokers.



Kafka APIs

- **Admin API** to manage and inspect topics, brokers, and other Kafka objects.
- **Producer API** to publish (write) a stream of events to one or more Kafka topics.
  `KafkaProducer<K,V> (implements Producer<K,V>)`
- **Consumer API** to subscribe to (read) one or more topics and to process the stream of events produced to them.
  `KafkaConsumer<K, V> (implements Consumer<K,V>)`
- The **Kafka Streams API** to implement stream processing applications and microservices.
  `KStream<K, V>`
  - To process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more.
- The **Kafka Connect API** to build and run import/export connectors to external systems and applications.
  - Kafka community already provides hundreds of ready-to-use connectors.

Event-driven patterns

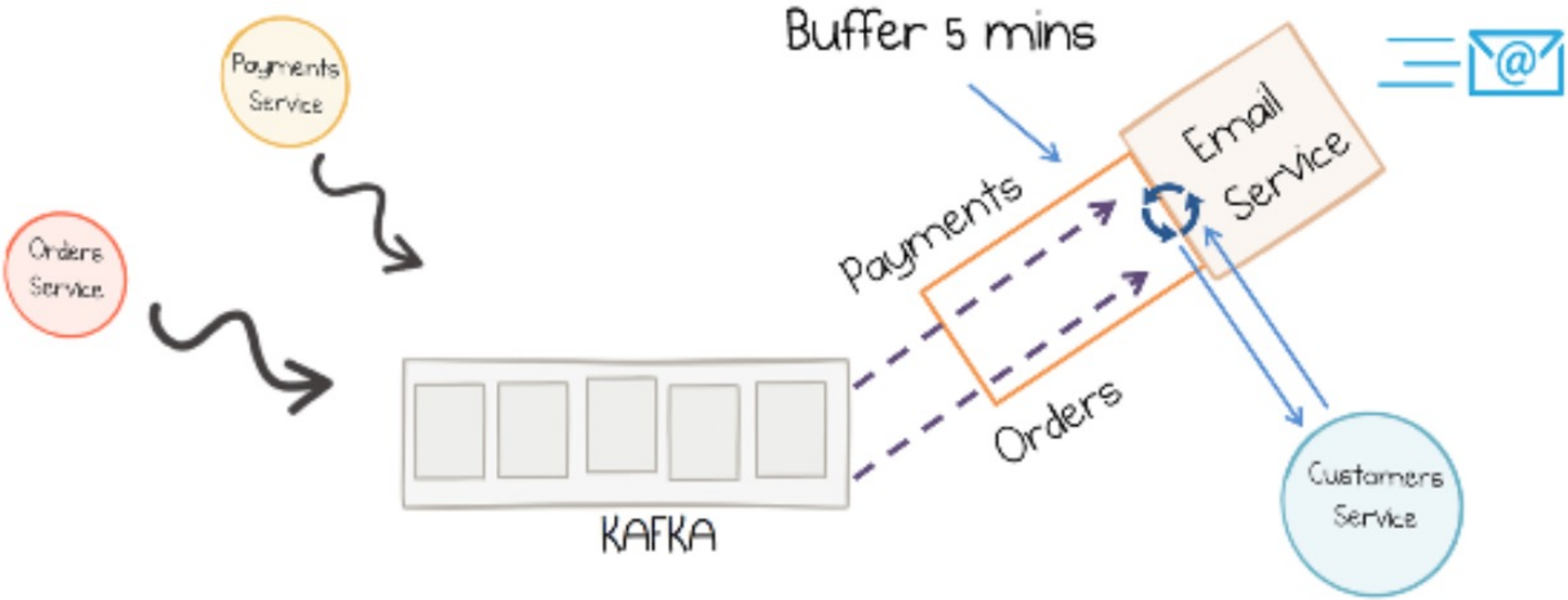
- Previously…
  - Event notification, Event-carried state transfer, Event-sourcing
- In **Event Sourcing**, events are a core element – the source of true.
  - Being stored, immutably, in the order they were created in, the event log expresses exactly what the system did.
- **Command Query Response Segregation** (**CQRS**) is a natural progression from this.
  - Decoupling writing and reading operations.
  - As a simple example, we might write events to Kafka (write model), read them back, and then push them into a data-base (read model).

Example

- Say we have an email service that listens to an event stream of orders and then sends confirmation emails to users once they complete a purchase.
  - This requires information about both the order as well as the associated payment. Such an email service might be created in several different ways

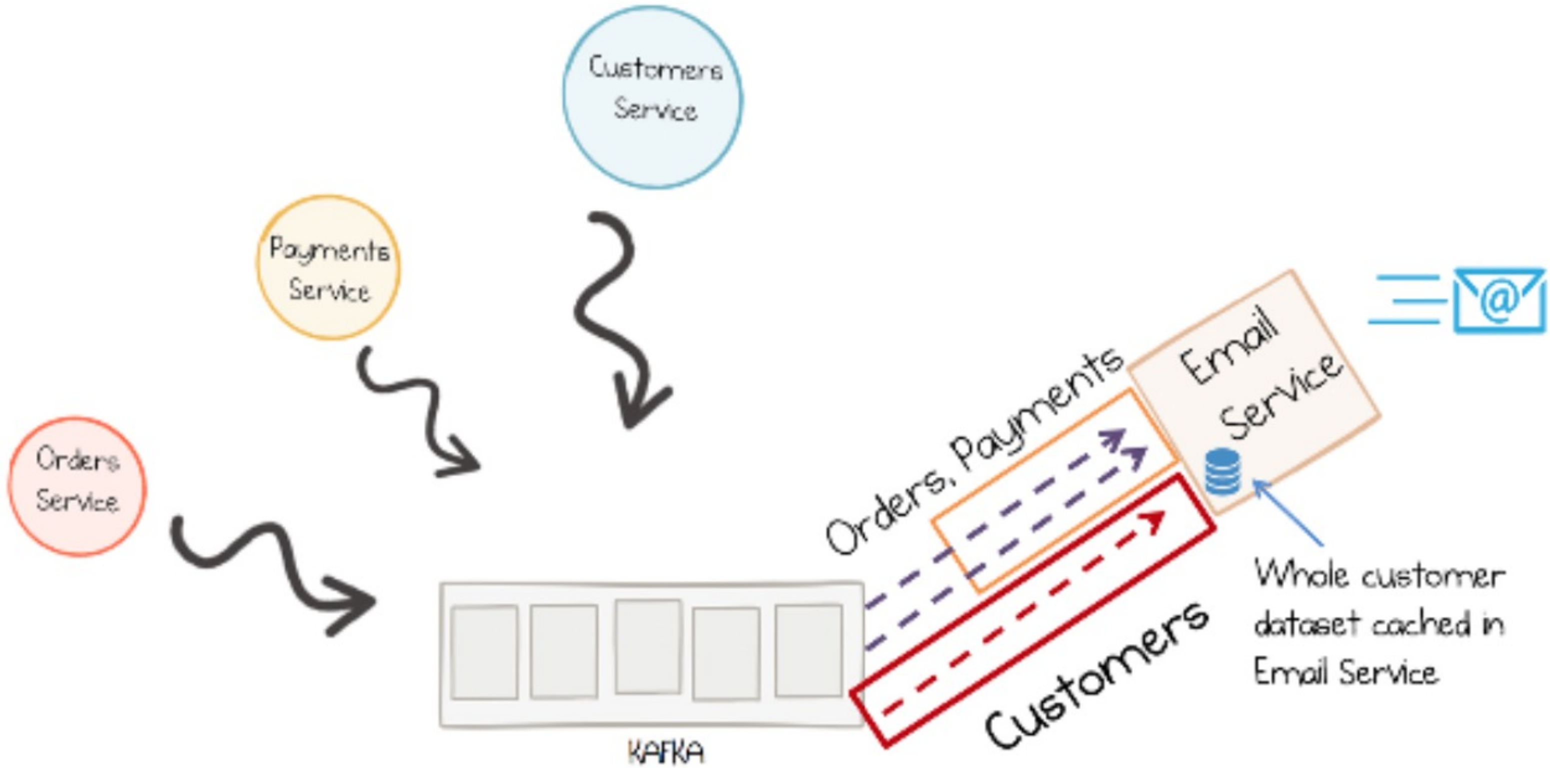
Stateless Streaming approach

❖ Example: A **stateless streaming** service that looks up reference data in another service at runtime



Stateful Streaming approach

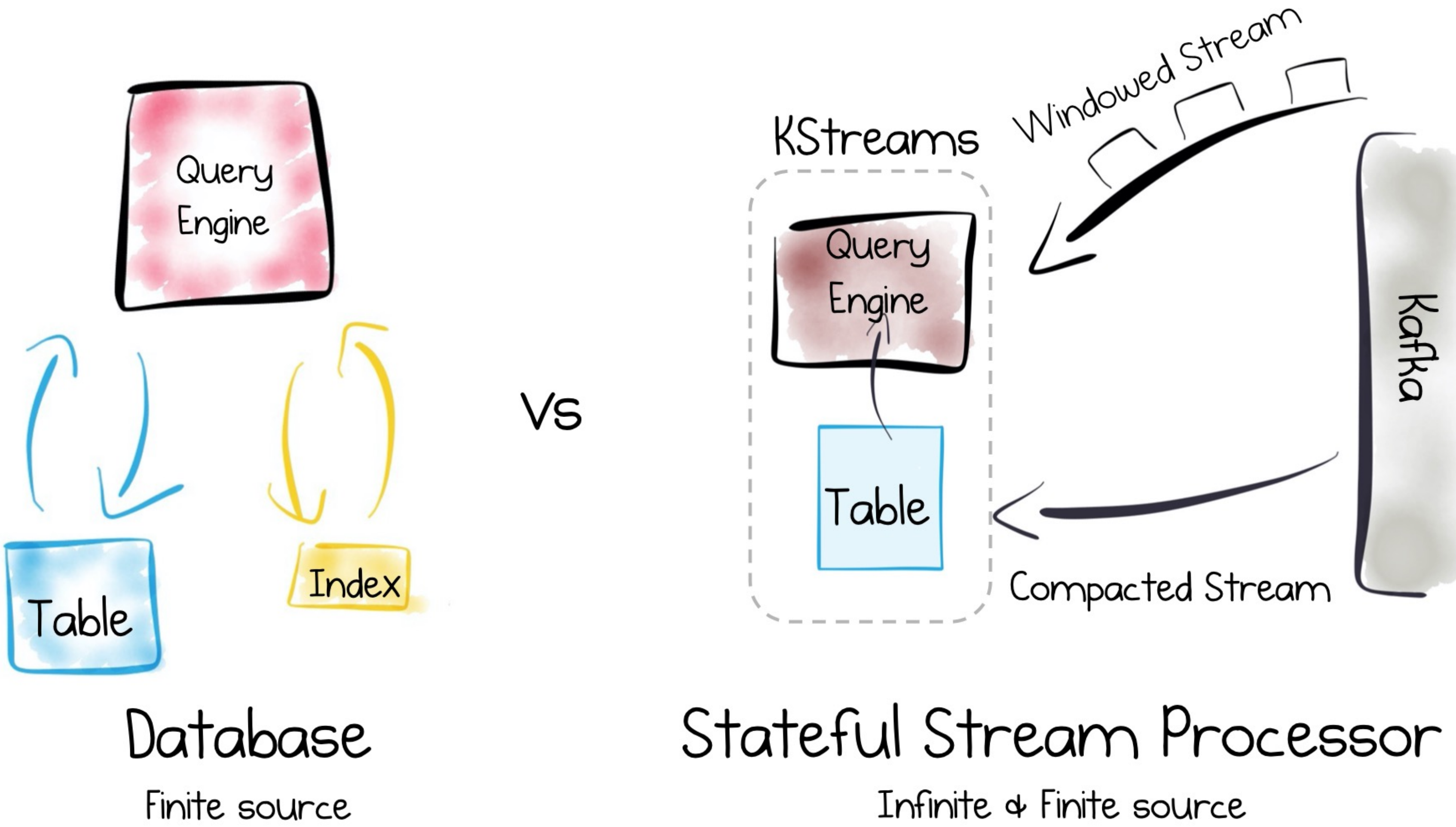
- Example: A **stateful streaming** service that replicates the Customers topic into a local table, held inside the Kafka Streams API.



Stateful Streaming approach

- Being stateful comes with some challenges:
  - when a new node starts, it must load all stateful components (i.e., state stores)
- **Kafka Streams**, for instance, provides three mechanisms to simplify stateful:
  - It uses a technique called **standby replicas**, which ensure that for every table or state store on one node, there is a replica kept up to date on another.
  - **Disk checkpoints** are created periodically so that, should a node fail and restart, it can load its previous checkpoint.
  - Finally, **compacted topics** are used to keep the dataset as small as possible. This acts to reduce the load time for a complete rebuild should one be necessary.

A Stateful Stream Processing



Messaging in Spring

- **Spring Cloud Stream** helps fully abstract code from the underlying messaging engine
- It supports a variety of binder implementations such as:
  - RabbitMQ
  - Apache Kafka
  - Kafka Streams
  - Amazon Kinesis
  - Google PubSub (partner maintained)
  - Solace PubSub+ (partner maintained)
  - Azure Event Hubs (partner maintained)
  - Apache RocketMQ (partner maintained)

Messaging in Spring

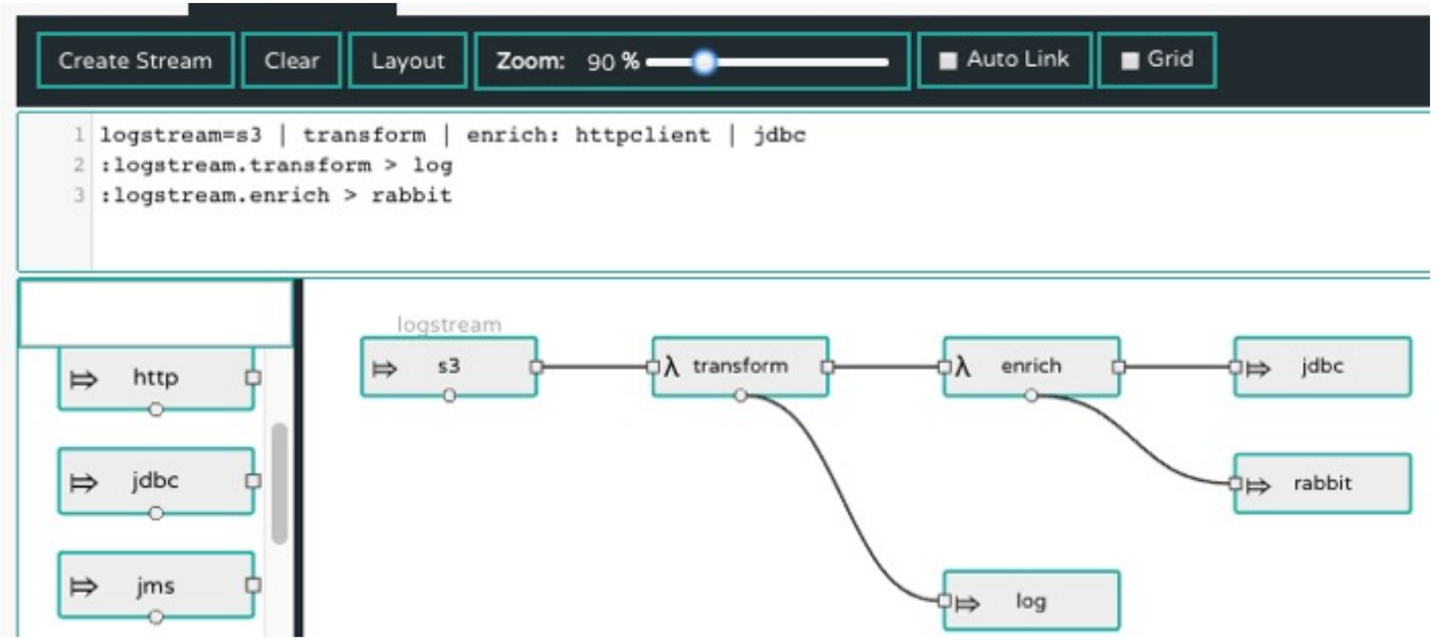
- **Spring AMQP**
  - It applies core Spring concepts to the development of AMQP-based messaging solutions.
  - The project consists of two parts; spring-amqp is the base abstraction, and spring-rabbit is the RabbitMQ implementation.

- **Spring for Apache Kafka**
  - Simple snippet

```
@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
public static class KafkaStreamsExampleConfiguration {
    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
        KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
        stream.map((k, v) -> new KeyValue<>(k, v.toUpperCase())).to("ks1Out",
                Produced.with(Serdes.Integer(), new JsonSerde<>()));
        return stream;
    }
}
```

Messaging in Spring

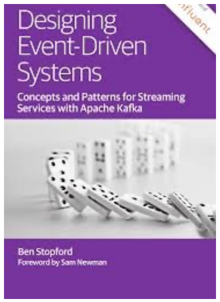
- **Spring Cloud Data Flow** allows to create and orchestrate data pipelines, e.g., data ingest, real-time analytics, and data import/export.

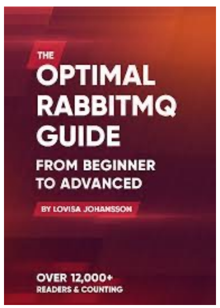


Summary

- Event-driven architecture is gaining in popularity, and with good reason.
  - From a technical perspective, it provides an effective method of wiring up microservices.
  - The interest in serverless functions - such as AWS Lambda, Azure Functions, or Knative - is growing, and these are inherently event-based.
  - Moreover, when coupled with modern streaming data tools like Apache Kafka, event-driven architectures become more versatile, resilient, and reliable than with earlier messaging methods.
- But perhaps the most important “feature” of the event-driven pattern is that it models how businesses operate in the real world.

Resources & Credits





- Designing Event-Driven Systems
  Ben Stopford, O'Reilly
- The Optimal RabbitMQ Guide, Lovisa Johansson, CloudAMQP
- Spring messaging projects
  - https://spring.io/projects/
- Kafka, RabbitMQ, etc..
  - https://kafka.apache.org
  - https://www.rabbitmq.com