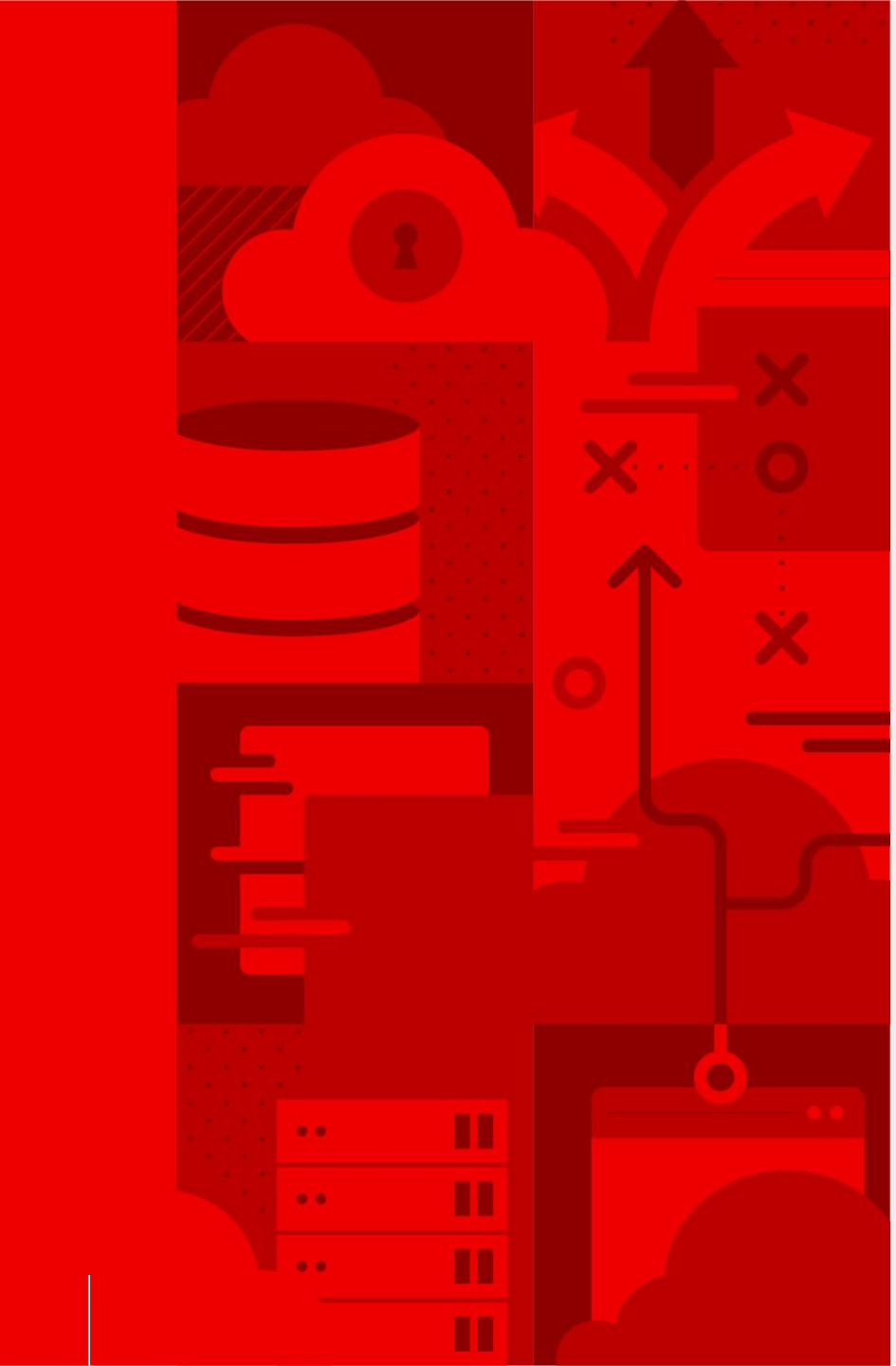


Modern event-driven Workloads with Knative

DevOpsCon 2023 - Munich

Roland Huß **@ro14nd**
Senior Principal Software Engineer, Red Hat
OpenShift Serverless Architect
Knative TOC alumni

Red Hat

Wait ... wat ?

Red Hat

Serverless

"Serverless computing refers to the concept of building and running **applications** that **do not require server management**. It describes a finer-grained **deployment model** where applications, bundled as one or more functions are uploaded to a platform and then **executed**, **scaled**, and **billed** in response to the exact **demand** needed at the moment"

-- CNCF Definition, https://www.cncf.io/blog/2018/02/14/cncf-takes-first-step-towards-serverless-computing/

Serverless vs. FaaS

Serverless is a **Deployment Model** that abstracts away the driving machine infrastructure.

- No server management required
- Executed, scaled and billed according to demand
- Defines a deployment packaging, but otherwise agnostic to the application

FaaS (Function-as-a-Service) is a **Programming Model** that mandates developing your application with fine grained function that match a given signature.

- Deployed as Serverless application
- Typically used as *glue code* to connect services


Knative

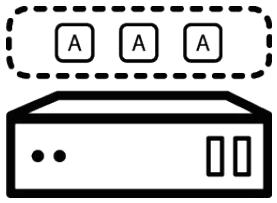
Kubernetes-based platform to **deploy** and **manage** modern **serverless workloads**.

https://knative.dev

Components

Serving

A request-driven model that serves the container with your application and can "scale to zero".



Eventing

Common infrastructure for consuming and producing events that will stimulate applications.

Function

Programming model that lets you focus on just your code for faster iterations.

Background Information

- Started as an **Open Source** Project mid-2018 by Google
- CNCF incubating project since 03/2022
- Community driven with a lot of vendor backing
  - https://github.com/knative
  - https://knative.dev
  - Support by Google, Red Hat, IBM, VMware, Triggermesh, SAP and more
  - Organized in multiple Working Groups with weekly meetings
- Releases
  - Current: **v1.10**
  - 4 releases / year cadence

Try Knative !

- Install from resource descriptors on Kubernetes Cluster
  - https://knative.dev/docs/install/
- Google **Cloud Run** (managed and on GKE)
  - https://cloud.google.com/run/
  - Not all Knative features implemented
    - see https://ahmet.im/blog/cloud-run-is-a-knative
- IBM Cloud **Code Engine**
  - https://www.ibm.com/cloud/code-engine
  - Vanilla Knative, additional workloads (like batch)
- Red Hat **OpenShift Serverless**
  - https://www.openshift.com/learn/topics/serverless
  - Supports all Knative features


Serving

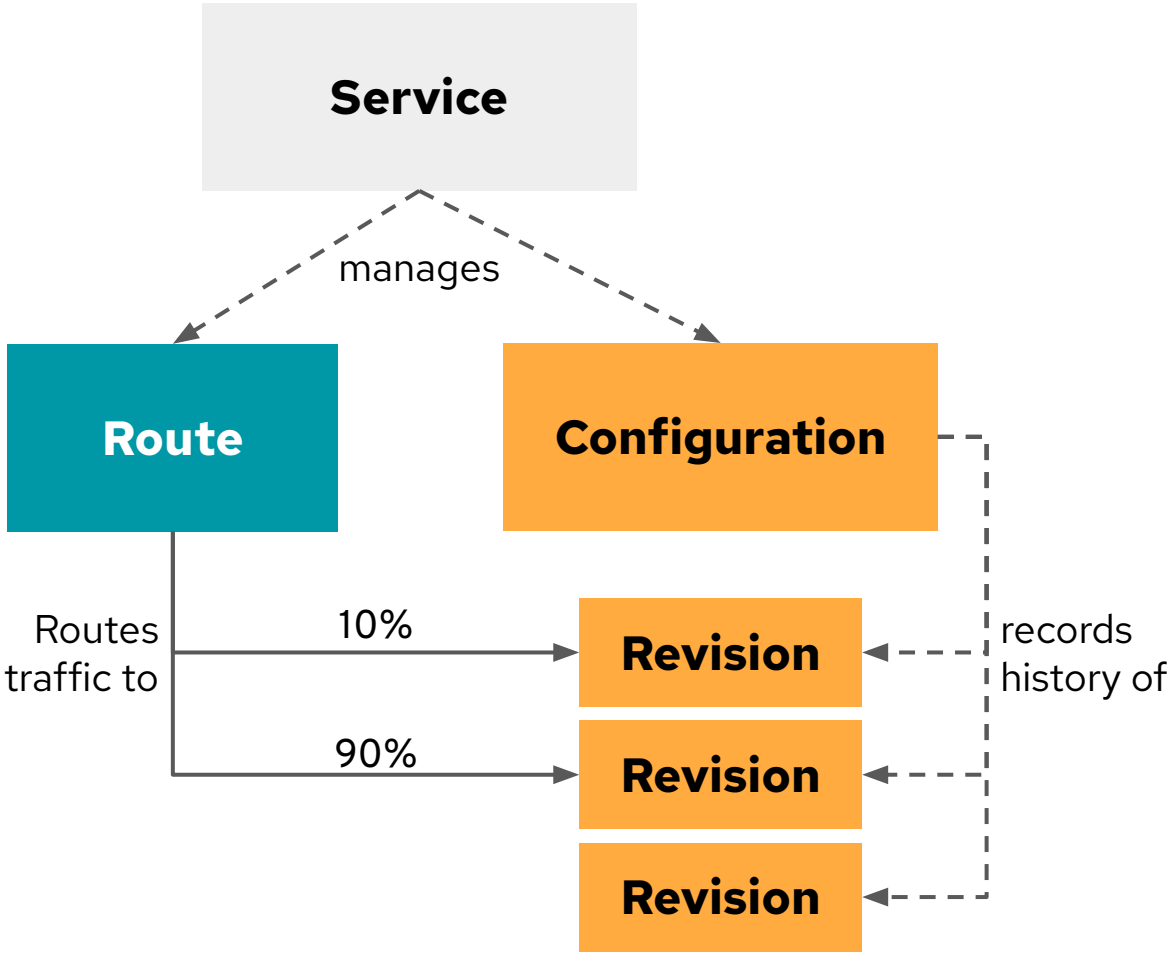
Route, **scale-to-zero** and track application **revisions** with ease.

Concepts

- **Demand-based autoscaling**, including scale-to-zero
- Separation of code and configuration
- Opinionated and simplified deployment model catered for **stateless applications**
- Rich **traffic split capabilities** to enable custom rollout strategies of new versions

Resources

- **Configuration** represent the *floating HEAD* of a history of **Revisions**
- **Revision** represents an immutable snapshot of code and configuration
- **Route** configure ingress over a collection of Revisions
- **Service** (not K8s services !) is a top-level entity that manage a set of Routes and Configurations



From **Deployment** to **KService**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random
  template:
    metadata:
      labels:
        app: random
    spec:
      containers:
      - image: rhuss/random
        name: random
        ports:
        - containerPort: 8080
```

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: random
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random
  template:
    metadata:
      labels:
        app: random
    spec:
      containers:
      - image: rhuss/random
        name: random
        ports:
        - containerPort: 8080
```

Red Hat

Demo


Eventing

Universal subscription, **delivery**, and management of **CloudEvents**.

Eventing

cloudevents

- Based on CloudEvents (CNCF Standard)
- Pluggable event transport via **Channels**
  - In-Memory
  - Apache Kafka
  - Nats
  - RabbitMQ
- Flexible routing of events from Sources to Sinks
  - **Source**: Adapter for integrating 3rd party systems and emitting CloudEvents
  - **Sink**: Addressable endpoint for CloudEvents (like a Knative Service)

Event Sources

- Integrating 3rd party systems with Knative
- More often "**Adapter**" than an original event source
- Declared with a **Custom Resource**
- Evaluated by an Operator
- Push or Pull based
- Converting custom event formats to **CloudEvents**

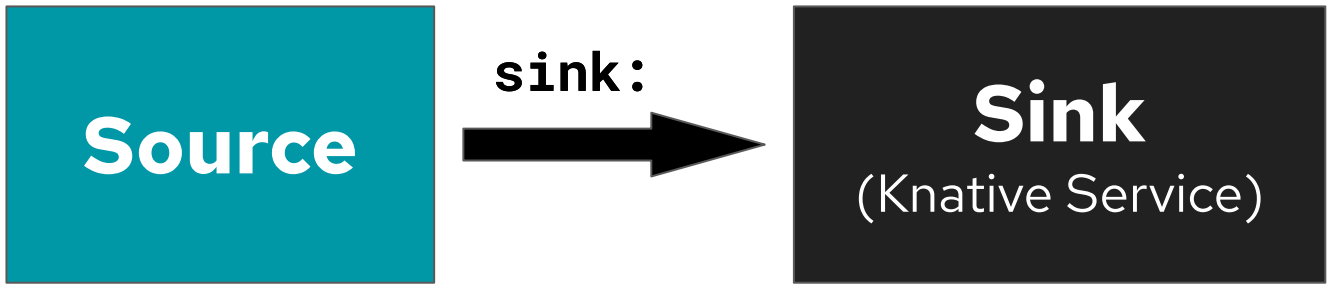
Sources

Builtin Sources	
PingSource	Emitting static CloudEvents periodically
ApiServerSource	Kubernetes API Server events as CloudEvents
SinkBinding	Binds an arbitrary Pod specification to a Sink
ContainerSource	Meta-Source combining SinkBinding & Deployment

Contributed Sources	
GitHubSource	Converts GitHub webhooks events to CloudEvents
KafkaSource	Apache Kafka messages as CloudEvents
RabbitMQSource	Import RabbitMQ messages as CloudEvents

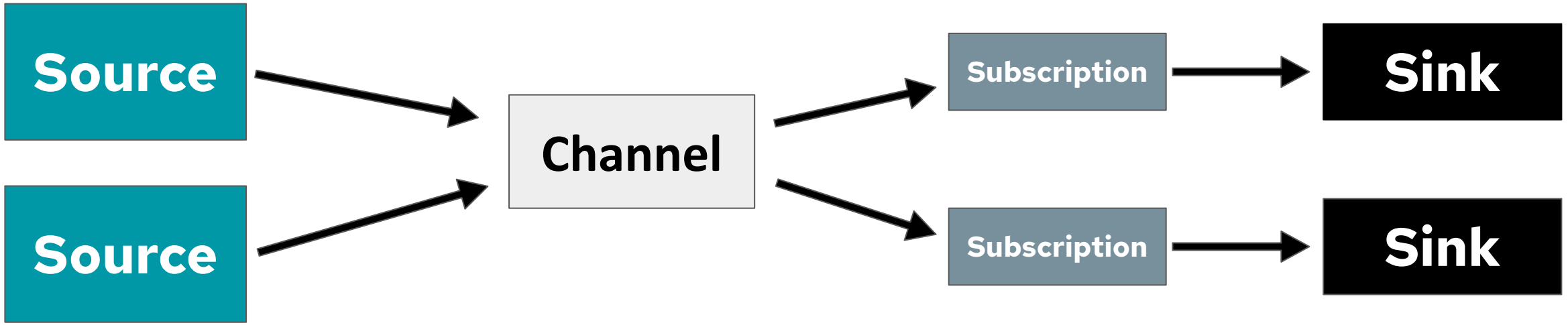
and many more: https://knative.dev/docs/eventing/sources/

Source → **Service** : Direct Connection



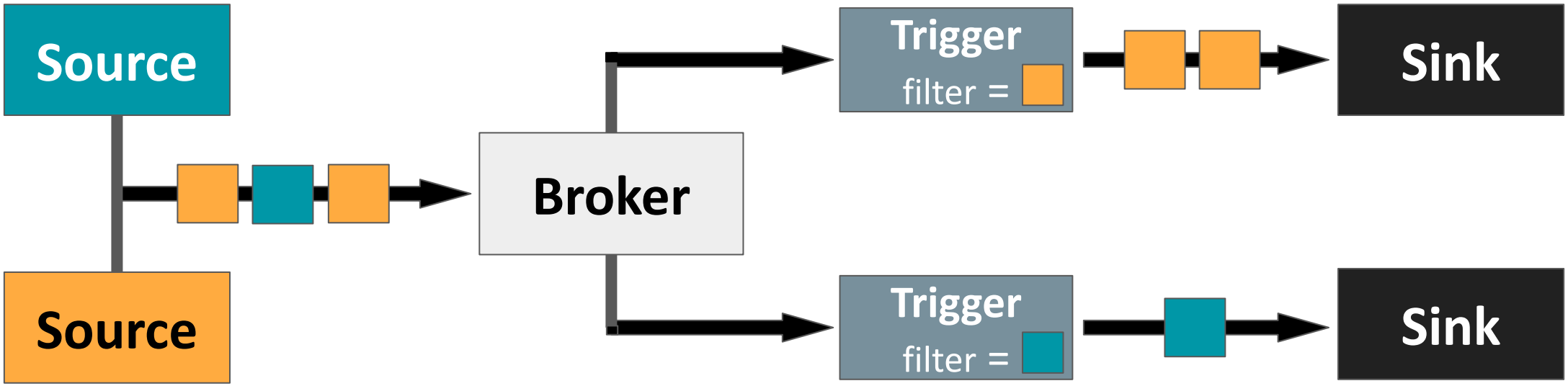
- Simplest way to get CloudEvents to a service
- Drawbacks:
  - No queuing support when service is unavailable
  - No back pressure support
  - Only one Service can consume events
  - No filtering, Service gets always all events

Source → Service : Channel & Subscription



- Multiple Services can consume the same event
- Subscription can point to a reply channel (not shown here)
- Various Channel Backends available
  - In-Memory, Kafka, RabbitMQ, NATS, (write your own)
- Drawbacks:
  - Channel Infrastructure needs to be set up manually
  - No filtering, Service gets always all events

Source → Service: Broker & Trigger



Broker

- Eventing Mesh for distributing Events
- Addressed by sources as sink

Trigger

- Filter on CloudEvent attributes (e.g. type)
- Connects a Sink with Broker

Broker

Broker API	
`Broker`	Brokers provide a well-known endpoint for event delivery that senders can use with minimal knowledge of the event routing strategy.

Implementation (via class annotation)	
`Channel Broker`	Reference implementation, based on Channel
`Kafka Broker`	Ingress/Egress of Apache Kafka records as CloudEvents (binary)
`Rabbit Broker`	Uses RabbitMQ for persistence

Source → Service: Broker & Trigger

- **Broker**
  - Eventing Mesh (or Event Delivery System)
  - Connects Sources with Sinks
  - Uses Channels internally, creating on the fly
  - Multi-tenant
- **Trigger**
  - Filter events (e.g. type and/or source)
  - Can produce new events (HTTP response is returned to Broker)
  - Delivered as CloudEvents
  - Dead-Letter Sinke for events that could not be processed

More Knative Eventing

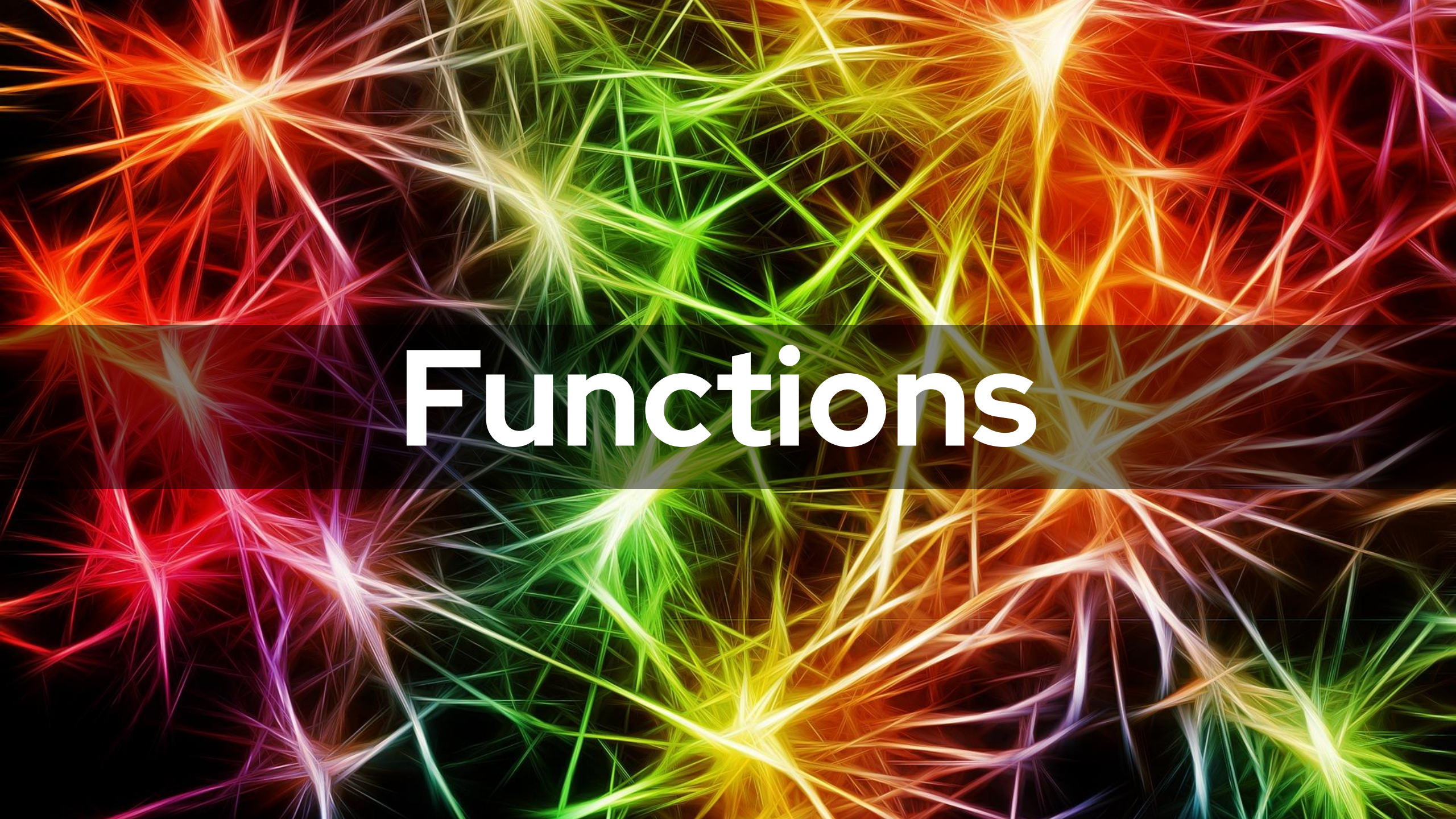
- **EventRegistry**
  - EventType CRD
  - Discoverability of Events
- **Sequence**
  - Chaining multiple Services
  - Sinking to an “Addressable” (Service, Channel, Sequence, Broker ...)
- **Parallel**
  - Branching of events with filters
  - Allows to implement conditional processing

Demo


Kamelets

Kamelets

- Part of **Camel-K**, a runtime platform on top of Kubernetes for Apache Camel routes
- Snippets that contain a **route template**
- Types of Kamelets
  - **Sources** for incoming cloud events
  - **Sinks** for outgoing cloud events
  - **Actions** for transformations
- Kamelets are instantiated via **KameletBindings**
  - Filling in Kamelet parameters, like authentication information
  - Connecting to sink (for sources)
- kn-source-kamelet : Kn **plugin** for managing Kamelets


Functions

kn-func

- Opinionated **programming model** for Knatve
  - Scaffolding of project templates for multiple runtimes
    - Quarkus, Node.js, Spring, Python, ...
  - Building and pushing container images
  - Deploying as Knative services
- Available as **plugin** of the Knative CLI
  - https://github.com/knative-sandbox/kn-func
- **Local development** mode
- Technologies
  - Cloud-native Buildpacks
  - Local Docker or Podman
  - On-cluster builds

Demo

Summary

Summary

Knative Serving

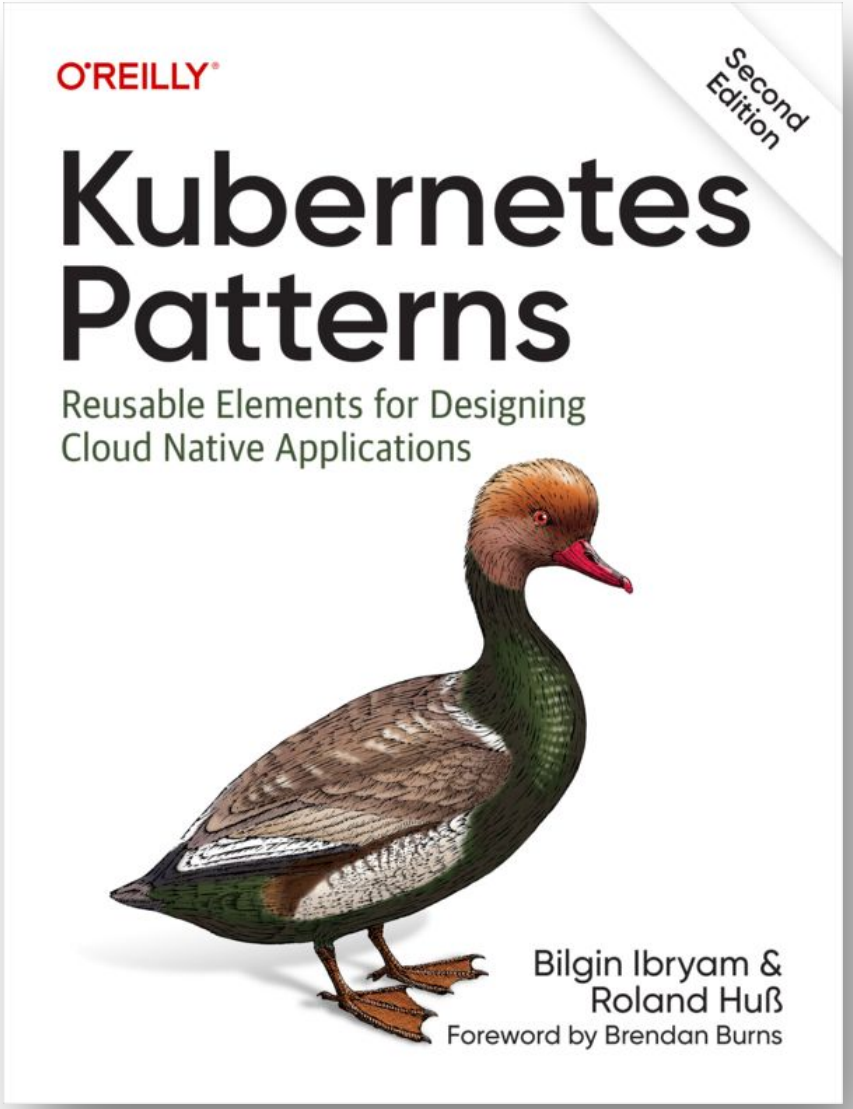
- Simplified Deployment for stateless workloads
- Traffic based autoscaling including Scale-to-Zero
- Traffic splitting for custom rollout / rollback scenarios

Knative Eventing

- External Triggers for feeding Knative Services
- Based on CloudEvents
- Backed by proven messaging systems
- Flexible messaging setup

http://dn.dev/knative-cookbook

Red Hat



https://k8spatterns.io

Red Hat

cncf.slack.com → #knative-48 – 6.12.2023 4pm CET to 8.12.2023 4pm CET

Thank you

ro14nd@hachyderm.io

Red Hat

Picture Credits

https://www.pexels.com/photo/boat-island-ocean-sea-218999/

https://unsplash.com/photos/t6t2-gXKxXM

https://unsplash.com/photos/UGMf30W28qc

https://pixabay.com/photos/hamburg-speicherstadt-channel-2976711/

https://pixabay.com/photos/beer-machine-alcohol-brewery-1513436/

https://pixabay.com/photos/camel-sunset-landscape-tourism-2500618/

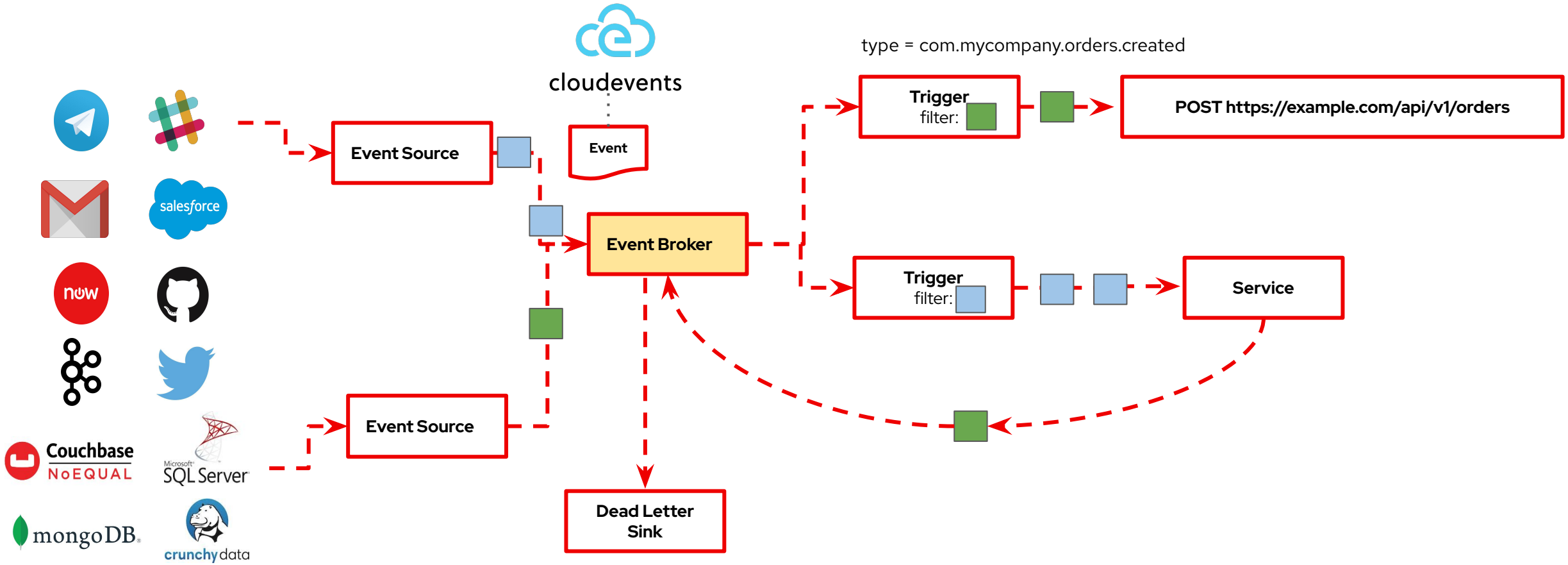
https://unsplash.com/photos/9SWHIgu8A8k

https://me.me/i/aws-lambda-is-just-glorified-cgi-bin-imgflip-com-change-my-mind-d0b715592ba34b08b79452ad02783ca2

https://unsplash.com/photos/dodn_0TESN0

https://pixabay.com/photos/annoy-cells-stars-dendrites-sepia-2926087/

Knative Event Mesh



Event Sources

ro14nd@mastodon.green
matzew@hachyderm.io

Red Hat

Event Processing

- Use **filter**(s) on Triggers for CloudEvent attributes
  - *Do not subscribe to all events* (especially **`type`**)

- Return **application specific** CloudEvents in response to the delivery
  - Success: return a new event (avoid filter loops on type)
  - Error: return new event for application level errors
    - provide context of failing payload, if possible
  - Use different types for return, to avoid filter loops

- Use **Dead-Letter-Sinks** to catch not-delivered events
  - there can be lot's of things going wrong

Event Processing

```java
@Funq
public CloudEvent<ValidOrder> function(CloudEvent<Order> newOrder) {

    // call to external system
    final ValidOrder vo = orderService.verify(newOrder.data());
    final LocalDate eta = vo.getDeliveryDate();

    // return application specific events:
    if (eta.isAfter(LocalDate.now().plusDays(30))) {
        return CloudEventBuilder.create()
                .specVersion("1.0")
                .type("delayed.order.created")
                .source("http://my-corp.com/orders")
                .subject("delayed")
                .id(UUID.randomUUID().toString())
                .build(vo);
    } else {
        return CloudEventBuilder.create()
                .specVersion("1.0")
                .type("order.created")
                .source("http://my-corp.com/orders")
                .subject("ontime")
                .id(UUID.randomUUID().toString())
                .build(vo);
    }
}
```

Event Processing

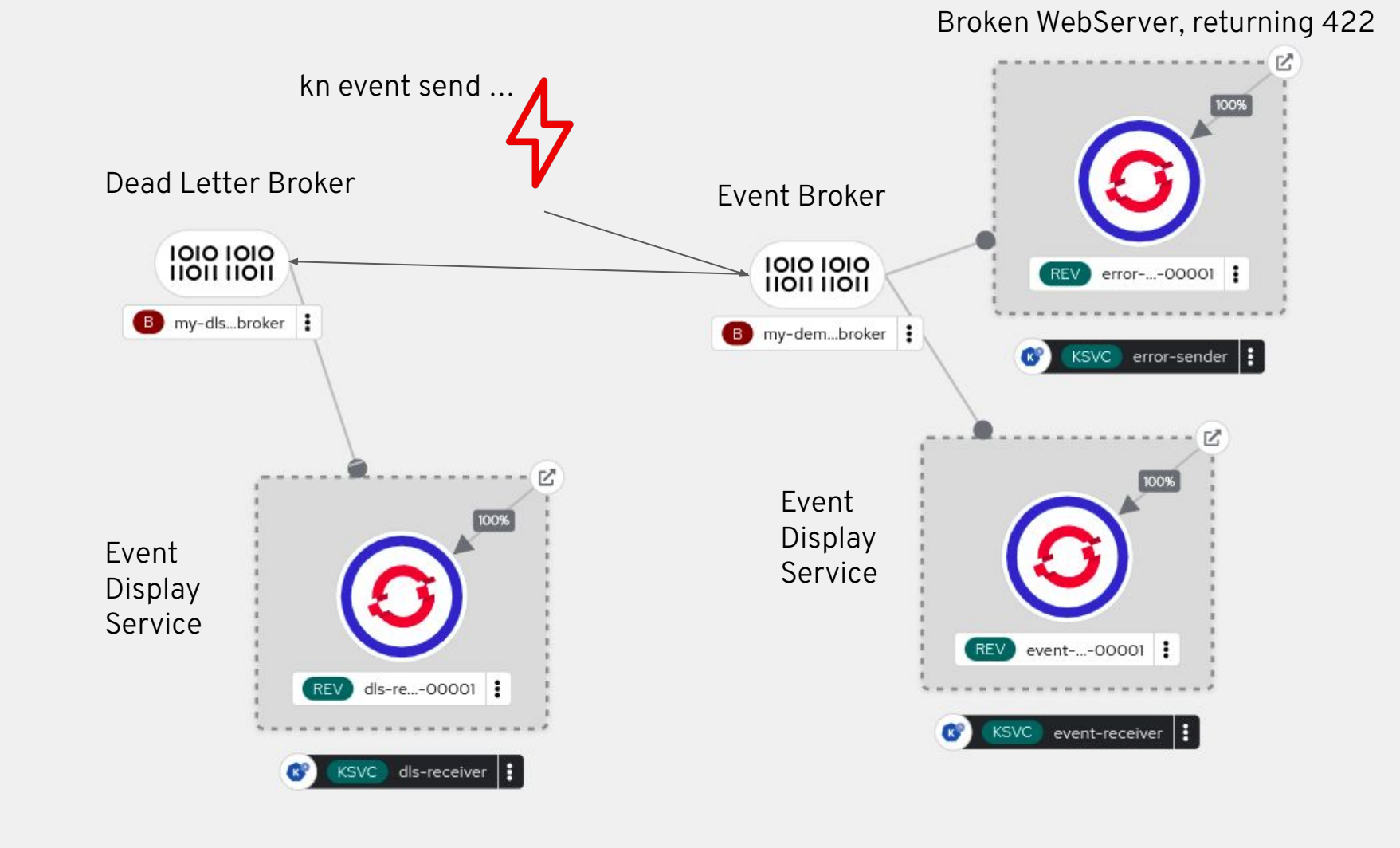
```
@Funq
public CloudEvent<ValidOrder> function(CloudEvent<Order> newOrder) {

    // call to external system
    final ValidOrder vo = orderService.verify(newOrder.data());

    // return a CloudEvent with the checked order
    return CloudEventBuilder.create().type(....)....build(vo);
}
```

Event Delivery Spec

- Concept of dead letter sink
- Retries
- Backoff
- Well defined retry behavior, based on HTTP status codes:

Response code	Meaning	Retry	Delivery completed	Error
`1xx`	(Unspecified)	No*	No*	Yes*
`200`	Accepted, event in reply	No	Yes	No
`202`	Event accepted	No	Yes	No
other `2xx`	(Unspecified)	No	Yes	No
`3xx`	(Unspecified)	No*	No*	Yes*
`400`	Unparsable event	No	No	Yes
`404`	Endpoint does not exist	Yes	No	Yes
`408`	Request Timeout	Yes	No	Yes
`409`	Conflict / Processing in progress	Yes	No	Yes
`429`	Too Many Requests / Overloaded	Yes	No	Yes
other `4xx`	Error	No	No	Yes
`5xx`	Error	Yes	No	Yes

Demo


Broken WebServer, returning 422
kn event send ...
Dead Letter Broker
Event Broker
B my-dls...broker
B my-dem...broker
100%
REV error-...-00001
KSVC error-sender
Event Display Service
100%
REV dls-re...-00001
KSVC dls-receiver
Event Display Service
100%
REV event-...-00001
KSVC event-receiver

Red Hat

(Kafka) Broker config

- Every broker has its own ConfigMap
- Setup a dedicated **broker** as the dead letter sink of the **broker**
  - Use Triggers to filter based on error code
- Override (delivery) defaults only when needed
- Kafka: 10 partitions and 3 replication nodes

New Trigger filters

- **CNCF Subscriptions API**
  - Experimental feature in Knative Eventing
    - Needs to be enabled via ConfigMap
  - Based on CloudEvent metadata
  - Supported dialects
    - Exact, prefix, suffix, all, any, not, cesql

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-suffix-trigger
spec:
  broker: default
  filters:
    - suffix:
        type: .created:
```


ro14nd@mastodon.green
matzew@hachyderm.io

New Trigger filters

- **CNCF Subscriptions API**
  - Experimental feature in Knative Eventing
    - Needs to be enabled via ConfigMap
  - Based on CloudEvent metadata
  - Supported dialects
    - Exact, prefix, suffix, all, any, not, cesql

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: little-bobby-tables
spec:
  broker: default
  filters:
    - cesql: "source LIKE '%commerce%' AND type IN ('order.created', 'order.updated', 'order.canceled')”
```


ro14nd@mastodon.green
matzew@hachyderm.io