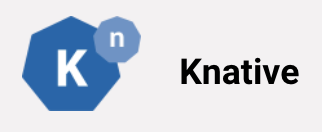
**Serverless Architecture with Knative**

Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications.

**Introduction to Knative**



* Knative is basically an open-source project which adds components for deploying, running, and managing serverless applications on Kubernetes.
* Developers can package their services or functions as a container image and hand it over to Knative. Knative then runs the container for a specific service only when it needs to.
* The core architecture of Knative comprises two broad components, Serving, and Eventing that run over an underlying Kubernetes infrastructure.
* Knative provides a set of building blocks that enable modern, source-centric and container-based development workloads on Kubernetes:
  + Build - Source-to-container build orchestration
  + Eventing- Management and delivery of events
  + Serving - Request-driven compute that can scale to zero



**Why We need Serverless Containers**

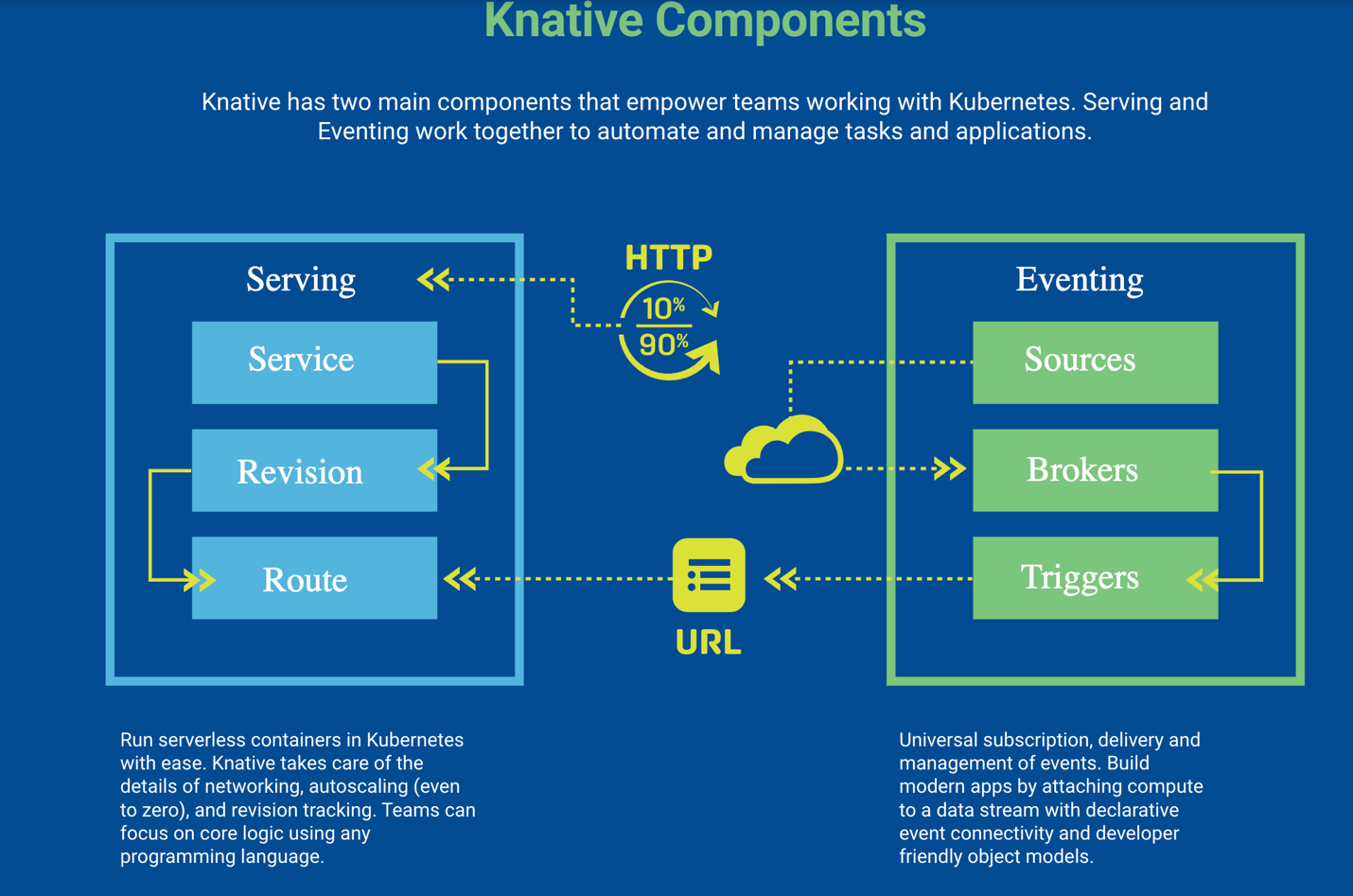
Serverless refers to running back-end programs and processes in the cloud. Serverless works on an as-used basis, meaning that companies only use what they pay for. Knative is a platform-agnostic solution for running serverless deployments.

**Benefits of Serverless Containers**

* Simpler Abstractions (Simplify your YAML with our custom CRDs - Custom Resource Definitions)
* Autoscaling (Scale down to zero and up from zero)
* Progressive Rollouts (Choose your rollout strategy depending on your needs e.g. Blue/Green and Canary Deployments )
* Event Integrations (Handle events from many sources - Multitenancy)
* Handle Events (Trigger handlers from the event broker)
* Plugable (Kubernetes native to be integrated and extended)

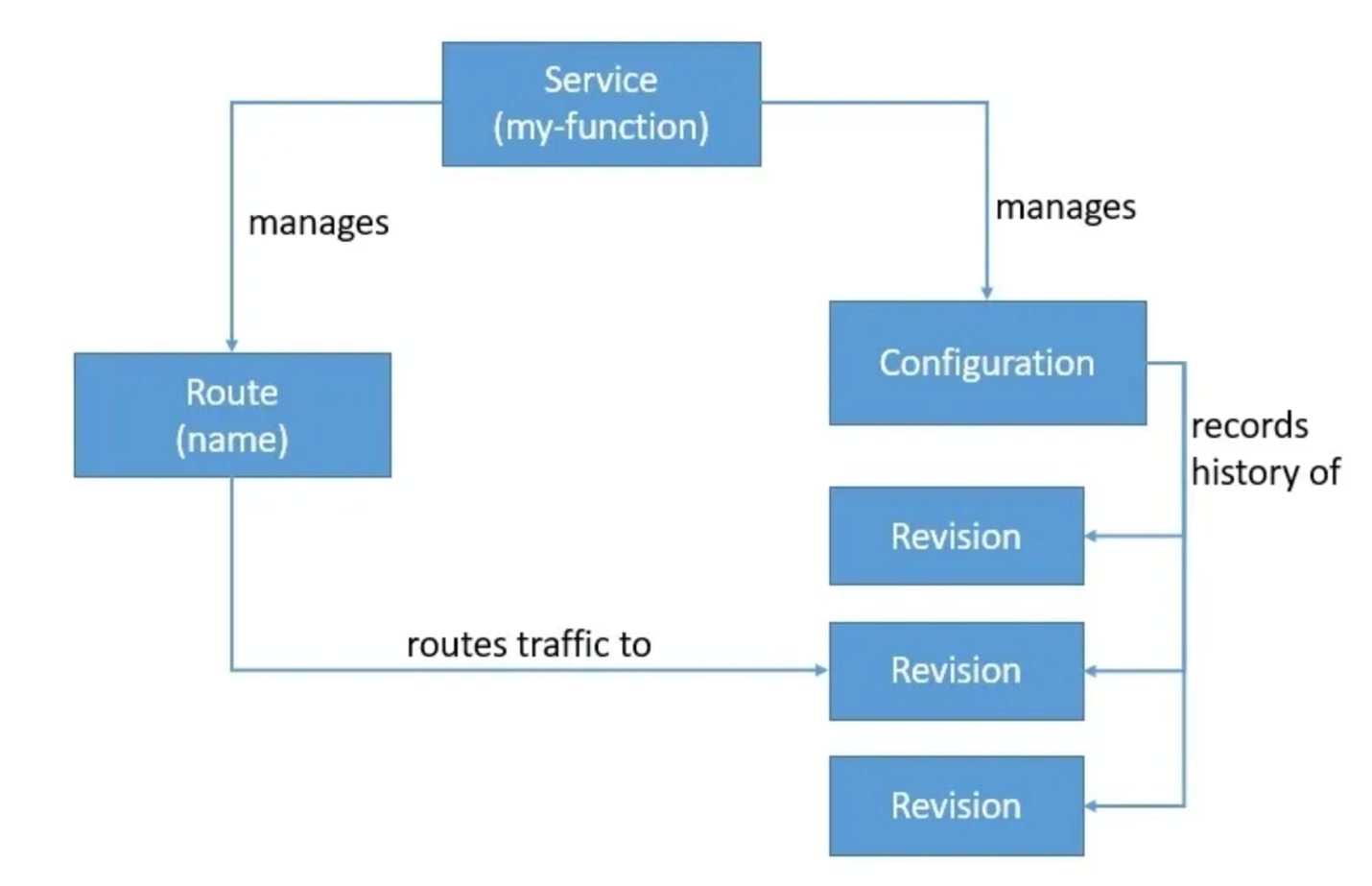
**KNative Components**

**Knative has two main components that empower teams working with Kubernetes. Serving and Eventing work together to automate and manage tasks and applications.**



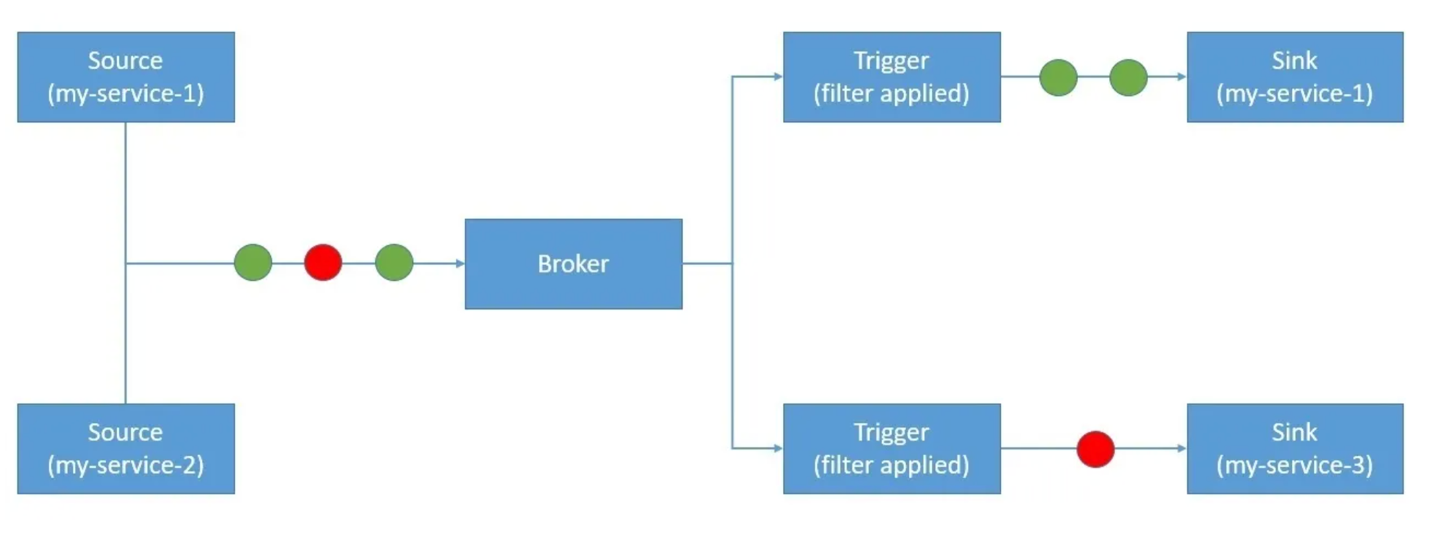
**Knative Serving**

* Knative Serving allows us to deploy containers that can scale automatically as required. It builds on top of Kubernetes and Istio by deploying a set of objects as Custom Resource Definitions (CRDs)
* Knative Serving primarily consists of four such objects, **Service, Route, Configuration, and Revision**. The Service object manages the whole lifecycle of our workload and automatically created other objects like Route and Configuration. Each time we update the Service, a new Revision is created. We can define the Service to route the traffic to the latest or any other Revision.
* **Service**: The service.serving.knative.dev resource automatically manages the whole lifecycle of your workload. It controls the creation of other objects to ensure that your app has a route, a configuration, and a new revision for each update of the service. Service can be defined to always route traffic to the latest revision or to a pinned revision.
* **Route**: The route.serving.knative.dev resource maps a network endpoint to one or more revisions. You can manage the traffic in several ways, including fractional traffic and named routes.
* **Configuration**: The configuration.serving.knative.dev resource maintains the desired state for your deployment. It provides a clean separation between code and configuration and follows the Twelve-Factor App methodology. Modifying a configuration creates a new revision.
* **Revision**: The revision.serving.knative.dev resource is a point-in-time snapshot of the code and configuration for each modification made to the workload. Revisions are immutable objects and can be retained for as long as useful. Knative Serving Revisions can be automatically scaled up and down according to incoming traffic. See Configuring the Autoscaler for more information.



**Knative Eventing**

* Knative Eventing provides an infrastructure for consuming and producing events for an application. This helps in combining event-driven architecture with a serverless application.
* Knative Eventing resources are loosely coupled, and can be developed and deployed independently of each other.
* Knative Eventing works with custom resources like **Source, Broker, Trigger, and Sink**. We can then filter and forward events to a subscriber using Trigger. Service is the component that emits events to the Broker. The Broker here acts as the hub for the events. We can filter these events based on any attribute using a Trigger and route then to a Sink.
* Knative Eventing uses HTTP POST requests to send and receive events conforming to the **CloudEvents**.
* CloudEvents is basically a specification for describing event data in a standard way. The objective is to simplify event declaration and delivery across services and platforms. This is a project under the CNCF Serverless Working Group.



**Knative Eventing Components**

An event-driven architecture is based on the concept of decoupled relationships between event producers that create events, and event consumers, or sinks, that receive events. It builds on delivery over HTTP by providing configuration and management of pluggable event-routing components.

A sink or subscriber can also be configured to respond to HTTP requests by sending a response event. Examples of sinks in a Knative Eventing deployment include Knative Services, Channels and Brokers.

**Event sources**

* In a Knative Eventing deployment, event Sources are the primary event producers. Events are sent to a sink or subscriber.

**Broker and Trigger**

* Brokers and Triggers provide an "event mesh" model, which allows an event producer to deliver events to a Broker, which then distributes them uniformly to consumers by using Triggers.

This delivers the following benefits:

* Consumers can register for specific types of events without needing to negotiate directly with event producers.
* Event routing can be optimized by the underlying platform using the specified filter conditions.

**Channel and Subscription**

* Channels and Subscriptions provide a "event pipe" model which transforms and routes events between Channels using Subscriptions.
* This model is appropriate for event pipelines where events from one system need to be transformed and then routed to another process.

**Event registry**

* Knative Eventing defines an EventType object to make it easier for consumers to ABC the types of events they can consume from Brokers.
* The registry consists of a collection of event types. The event types stored in the registry contain (all) the required information for a consumer to create a Trigger without resorting to some other out-of-band mechanism.

**Knative Usecases**

* **Traffic Splitting Using Knative Serving**
  + Scaling a serverless workload up and down automatically is not the only benefit of using Kantive Serving. It comes with a lot of other power-packed features that make the management of serverless applications even easier.
  + If we recall the concept of Revision in Knative Serving, it's worth noting that by default Knative directs all the traffic to the latest Revision. But since we still have all the previous Revisions available, it's quite possible to direct certain or all traffic to an older Revision.All we need to do to achieve this is to modify the YAML file that had the description of the Service:

1apiVersion: serving.knative.dev/v1

2kind: Service

3metadata:

4 name: my-service

5spec:

6 template:

7 metadata:

8 name: my-service-v2

9 spec:

10 containers:

11 - image: <location\_of\_container\_image\_in\_a\_registry>

12 ports:

13 - containerPort: 8080

14 traffic:

15 - latestRevision: **true**

16 percent: 50

17 - revisionName: my-service-v1

18 percent: 50

As we can see, we've added a new section (traffic) that describes the division of traffic between the Revisions. We are asking **Knative to send half the traffic to the new Revision while the other to the previous Revision**. After we deploy this resource, we can verify the split by listing all the Revisions:

1kn revisions list

* **Filter and Subscribe to Events Using Knative Eventing**
  + Knative Eventing helps us blend event-driven programming into the serverless architecture. But why should we care about event-driven architecture? Basically, **event-driven architecture is a software architecture paradigm that promotes the production, detection, consumption of, and reaction to events**.
  + An event-driven architecture is quite flexible and can be as simple as a single service to a complex mesh of hundreds of services. Knative Eventing provides the underlying infrastructure without imposing any restrictions on how we architect our applications.
  + Able to filter and send the events to specific targets. For this, we need to define a Trigger. Basically, Brokers use Triggers to forward events to the correct consumers. Now, in the process, we can also filter the events we want to send based on any of the event attributes.

1apiVersion: eventing.knative.dev/v1

2kind: Trigger

3metadata:

4 name: my-trigger

5 annotations:

6 knative-eventing-injection: enabled

7spec:

8 broker: <name\_of\_the\_broker\_as\_provided\_by\_borker\_list\_command>

9 filter:

10 attributes:

11 type: <my\_event\_type>

12 subscriber:

13 ref:

14 apiVersion: serving.knative.dev/v1

15 kind: Service

16 name: my-service

This is a quite simple Trigger that defines the same Service that we used as the Source as the Sink of the events as well. Interestingly, we are using a filter in this trigger to only send events of a particular type to the subscriber.

**Benefits and Challenges of Knative**

**Benefits**

* Knative has a number of benefits. Like OpenFaaS, Knative allows you to create serverless environments using containers. This in turn allows you to get a local event-based architecture in which there are no restrictions imposed by public cloud services.
* Knative also lets you automate the container assembly process, which provides automatic scaling. Because of this, the capacity for serverless functions is based on predefined threshold values and event-processing mechanisms.
* In addition, Knative, allows you to create applications internally, in the cloud, or in a third-party data center. This means that you are not tied to any one cloud provider. And due to its operation being based on Kubernetes and Istio, Knative has a higher adoption rate and greater adoption potential.

**Drawback**

* One main drawback of Knative is the need to independently manage container infrastructure.
* Simply put, Knative is not aimed at end users. However, because of this, more commercially managed Knative offers are becoming available, such as the Google Kubernetes Engine and Managed Knative for the IBM Cloud Kubernetes Service.

**References**

* Knative Github - <https://github.com/knative>
* Knative Code Samples - <https://knative.dev/docs/samples/>