RED HAT SUMMIT

Deliver cloud-native microservices with Knative on Red Hat OpenShift

Michael Costello and David Gordon

Boston - 5.8.2019

AGENDA

- **PLEASE** follow the following instructions:

  https://github.com/mike-costello/knative-build-samples/blob/master/README.md

- While you're doing that...
  - Let's talk about the journey to "Cloud Native"
  - By the time we've had our little talk, your environment should be ready to go and we'll walk through Knative together
    - We'll take a very generic java based "legacy" microservice and create a build pipeline
    - We'll take those builds, and run them serverlessly
  - If we have time, we'll walk through Knative Eventing

WHAT IS CLOUD NATIVE?

CLOUD NATIVE
COMPUTING FOUNDATION

Cloud native technologies **empower organizations to build and run scalable applications in modern, dynamic environments** such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

To The Cloud!

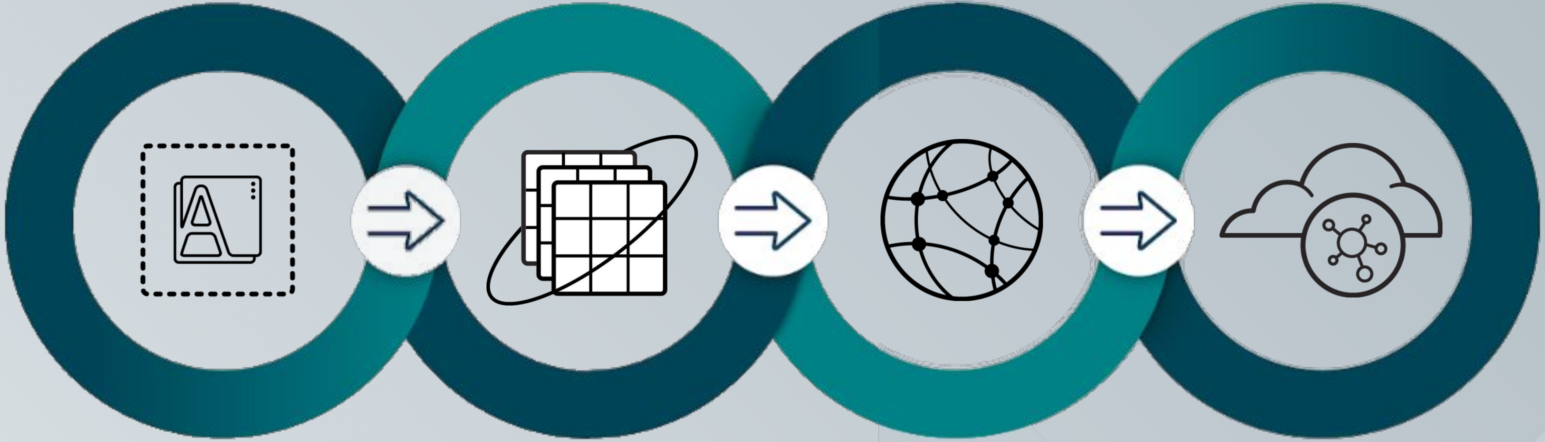
As we began to move our compute to the cloud, we noticed a few things and were made a few promises:

- Our traditional legacy ***monoliths didn't fit this space well***
  - Large bulky monoliths that we used to spend tons of time caring and feeding, didn't seem to fit well in a space where compute was infinitely scalable and likely ephemeral
  - We needed to do something more than simply "lift and shift" our monoliths to take advantage of cloud architecture
- Immutable, Scaled deployments would ***enhance reliability***
  - As the number of deployments increased, more and more things became more and more **difficult to manage**
  - Patterns began to emerge (such as service dependency graphs) that would imperil our runtimes altogether, we seemed to have more of a house of cards and less of a reliable platform
- Things like ***security, monitoring, visibility would get better.*** Unfortunately, they got ***worse***
- We would take on ***new architectural techniques*** that lent themselves more naturally to cloud architecture, well…they also created a ***new set of problems***

This imperils our road to the Cloud

- **"MicroService Architecture will enhance reliability and time to market"** Is that really working out?
  - Points of decomposition can seem arbitrary as decomposition becomes infinitely regressive and teams fight to avoid incurring **massive overhead** during this process.
  - All this decomposition means running **processes that aren't doing any actual work**. As we decompose more and more, more and more overhead attaches to our deployments. We also have a hard time continuing to sell that "the cloud is going to be cheaper"
- Service Dependency Graphs begin to emerge.
  - As Service A begins to depend on Services B,C, and D, and each of those services have their own set of dependencies, an outage in a single bit of compute can (and often has) cascade into **total outage**.
- Container Platforms, PAAS, and Managed Services would increase reliability and time to market, right?
  - Container platforms, PAAS, and Managed Services offer us abstraction points away from our infrastructure, but, don't ensure what we're actually putting into these things works well in a cloud environment. We need an **abstraction** away from the management of our applications.

Making it Better: Plotting a course toward **CLOUD NATIVE**



Container Adoption

Microservices Adoption

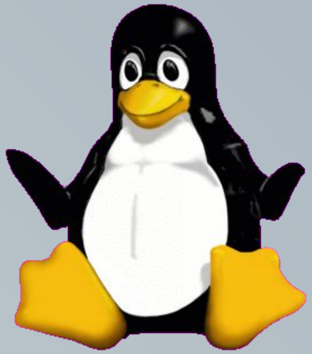
Service Mesh Adoption

Serverless Adoption

“Cloud Native” Maturity *- is my journey*

cloud native?

- As we moved these bits to the cloud and began our maturation journey, we typically:
  - Brought over all our old ***technical debt*** (and we had alot of little gremlins begin to peak out that were hidden before)
    - Means of ***managing identity*** (think keys/auth/auz) became ***nearly impossible*** as our number of ***deployments exploded.*** This was something that was always carefully manicured, and now it was the Wild Wild West
    - Our build and ***CICD pipelines***, that were always a source of concern for devops initiatives in our old crusty DC (but weren’t too big a deal), came ***front and center*** as more and more deployments relied on more and more things to happen all at once. What was “just good enough” was now **brittle and frail**
  - As our monoliths decomposed into more and more deployments, each carried the ***overhead*** they would have shared before. This ***“decomposition”*** thing began to seem like an ***anti-pattern***.

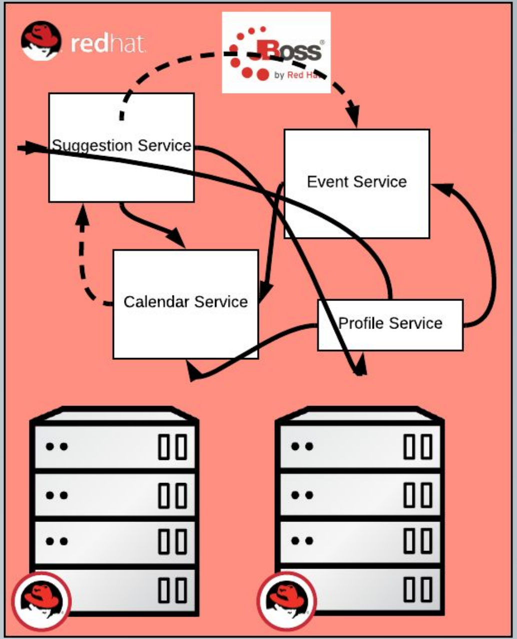


Acute DEVOPS Pain Points During Our “Cloud Native” Transition

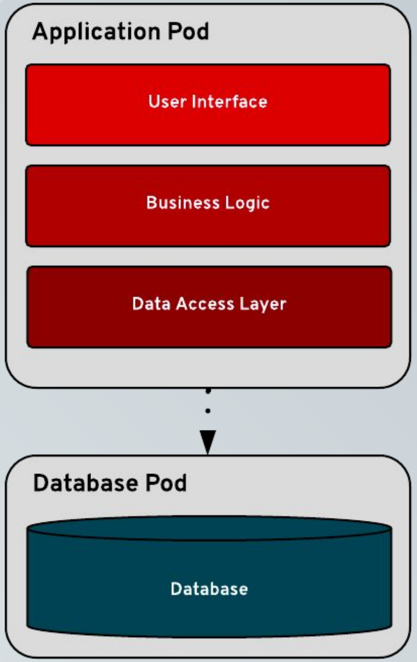
- “Legacy” Monolith decomposition in a ***number of different technologies*** - Java, C#, Perl, PHP
- Dockerfiles built by ***daemons running*** on cluster hosts
- <u>***Hundreds of Jenkins pipelines***</u> leveraging a diverse set of build, security, and testing tools
- Jenkins managing tons of OpenShift resources and ***orchestrating configuration changes***
- Constantly tweaking resource allocation configs to keep baseline cpu/memory footprint reasonable, and never really getting there
- ***Managed Services*** are attractive and all that but they’re only ***“native” to the cloud I’m in***

Impacts of *Decomposition*

- Dev teams are more independent, but…
  - ***CI/CD onboarding*** is still a ***bottleneck***
  - Devs want to ***decompose more than seems reasonable***
  - Devs want PaaS or FaaS, not just IaaS
- All this decomposition means running **processes that aren't doing any actual work**
- Instrumentation for observability and resilience is daunting
- Many, sometimes nested, synchronous REST calls forming **brittle service graphs**



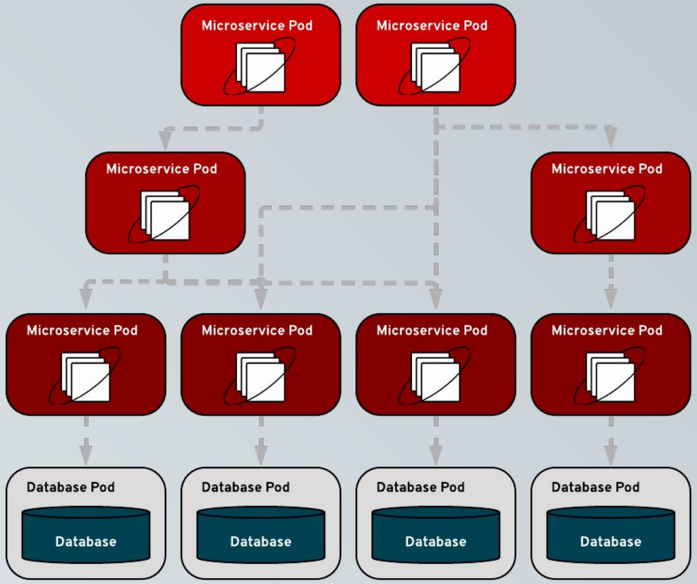
CONTAINER ADOPTION CHALLENGES



The Containerized Monolith

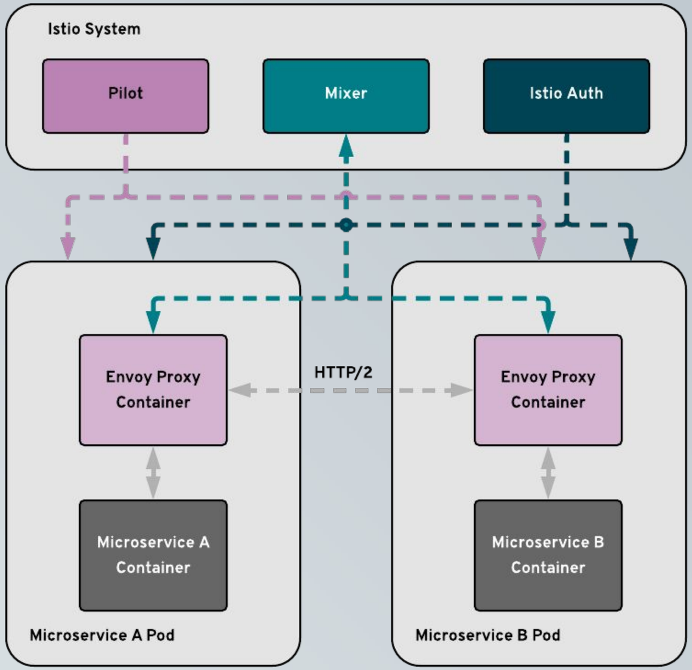
- **Heavy containers**
  - High CPU/Memory
  - Slow start times
- **Delivery is still slow**
  - Big code merges
  - Heavy release planning
- **All-or-nothing component resource scaling**
- **Limited resilience**

MicroService Architecture is CHALLENGING at Scale



- ***Observability***: fine-grained and aggregated metrics, tracing, and logs
- Traffic **routing** for multiple types of TCP
- Consistent approach to communication resilience e.g retries, circuit breakers
- Network policy control e.g. **rate limiting**
- Service-to-service **authorization**
- ***Runtime overhead*** of running many pods
- Complex CI/CD
- Each microservice requires **a lot of instrumentation**

A Service Mesh Solves This...right?



Istio helps a lot, but...

- Running all the system components Incurs ***even more resource overhead***
- Devs need to understand and configure ***low-level resources***
  - Ingress and egress
  - Traffic routing rules
- Would be nice to have developer-facing resource definitions that ***abstract service mesh details***
- Ingress/Egress, traffic routing rules, and how services interact should be ***baked into service deployment***

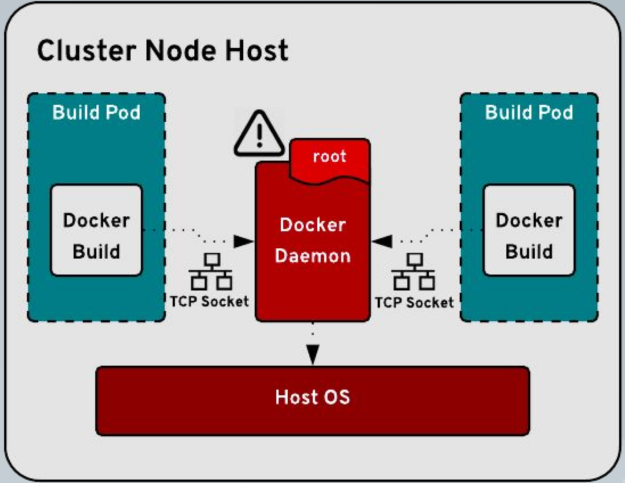
Brittle Pipeline Instrumentation *Struggles* at Scale

- Centralized master node is a ***single point of failure***
- Plugin dependency **hell**
- Streaming workspace and pipeline state data between pods
- Need special Jenkins build images for each diverse build step
  - Mixture of supportabillity and security concerns
  - Opaque dependency graphs and breaking change

Jenkins And A Docker Daemon

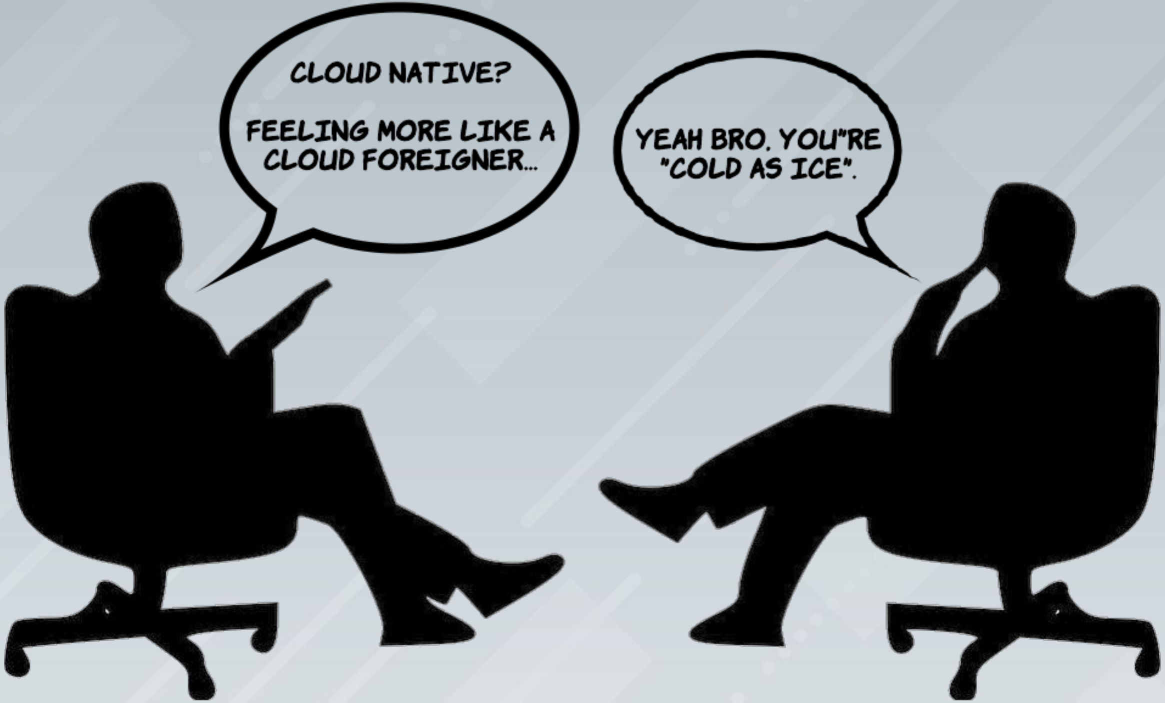
“The Horror!”

- Build containers ***require a socket to the daemon*** running on the host
- Limited authorization capabilities
- **<u>Clients have ‘root’</u>** on the host OS
- Daemon is shared across all containers running a build on the same cluster node
- **<u>All we want to do is build</u>**, not run!



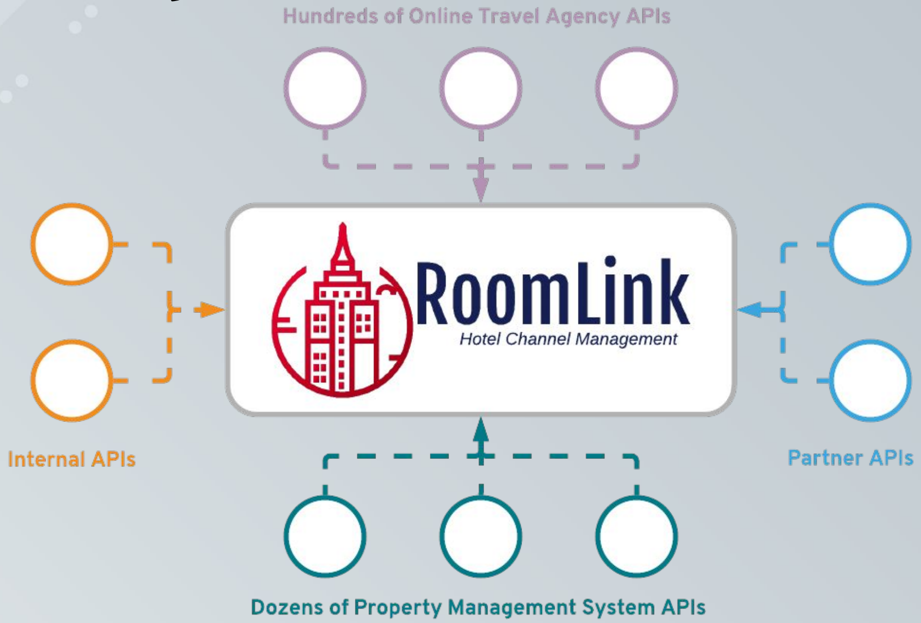
K8s/OPENSHIFT Deployments *(don’t quite get us all the way “there”)*

- **Can’t scale to zero**
  - Bloated compute infrastructure
  - Architectural concerns run afoul of the sheer realities of resource availability
  - Constant health checks consume compute
- **Can’t scale according to request demand**
  - Resource utilization sits idle
- Can’t **route** traffic to **multiple service revisions** native to my service implementation (I’ve gotta go do some network layer fun)
- ***SDLC orchestration*** becomes **<u>complex</u>** as we attempt to iterate over new features


CLOUD NATIVE?

FEELING MORE LIKE A CLOUD FOREIGNER...
YEAH BRO. YOU"RE "COLD AS ICE".

The Strawman: A Traditional Enterprise Integration Use Case


Hundreds of Online Travel Agency APIs
RoomLink
Hotel Channel Management
Internal APIs
Partner APIs
Dozens of Property Management System APIs

Integration Hub as a Service

- 1000s of API integrations
- API “adapters” as microservices
- Traffic bursts at certain times of the day and/or seasonally
- Need guaranteed message delivery
- Diverse legacy implementations migrated to containers
- Experimentation with modern runtimes and integration frameworks

What we need to be ***CLOUD NATIVE***

- ***Observability*** - tracing, metrics, logs
- ***Control points*** - traffic routing, security and authorization
- ***Loose coupling*** - focus on open specifications
- ***DevOps abstractions*** - APIs for developers
- ***Resource efficiency*** - serverless, request-driven autoscale to zero, and source to url idioms
- **Event-driven architecture** - channel binding abstractions

We need a new platform to ***EXTEND THE K8S PRIMITIVES*** we use today

What about decomposition?

- To be truly “cloud native” we actually **DO** need to follow our paths of decomposition
  - The ***“bounded context” is murky*** we need to **<u>go further</u>**
  - Decomposition to our **smallest meaningful unit**
    - **Decompose to the “function”**
    - **Coarse grained service composition** should emerge from our decomposition. “Without Integration MicroServices are meaningless” - Some smart guy we know ca. 2017
    - Our service endpoints, should emerge into a ***graph of enterprise behaviour(s)***

INTRODUCING KNATIVE

HISTORY AND COMMUNITY

- Google first announced the release of Knative in July 2018 in partnership with Pivotal, IBM, Red Hat and SAP.
- Open source under Apache License 2.0
- The latest release of Knative is version 0.5.

KNATIVE GOALS

Build - Provide a standard, portable, reusable, and performance optimized method for defining and running on-cluster container image builds

Serving - Provide a common toolkit and API framework for serverless workloads

Eventing - Define common, composable primitives to enable late-binding event sources and event consumers

Knative System Components

Knative Build

- Extends K8s Primitives For K8s Native Build Pipeline Orchestration
- On-Cluster Builds From Source
- Framework for Sequential Step Based Build Pipeline(s)

Knative Serving

- Built on top of Kubernetes and Istio
- Serverless deployments of applications and Functions
- Extends K8s primitives
- Scale to Zero
- Revision Based Deployment
- Routing and network programming for Istio components

Knative Eventing

- Event System to provide Channels, Triggers, Subscriptions and event source behaviour for loosely coupled producers and consumers
- Current support for disparate event sources including In-Memory, K8s Events, Apache Camel, Kafka and more...
- Declarative Primitives to Support event control plane

Additional Knative Components

Configuration

- Describes the desired **latest Revision state**
- Creates and tracks the **status** of Revisions
- Controller tracks status of created Revisions and makes the most recently created and most recently ***ready*** Revisions

Eventing Sources

- Provides framework for developing a source to deliver messaging events to an Addressable target.
- Number of event sources to provide channel based implementations for subscribers, triggers
- Exploits Knative Serving services as sinks for event sources

Knative Monitoring

- Evolving specification for monitoring capabilities that are native to Knative's systems
- Leverages Prometheus and other operators to instrument visibility across severless, and eventing components

KNATIVE BUILD

KNATIVE BUILD

- Declarative build semantics deployed as Kubernetes primitives
- Native hooks - leverages admission webhooks for automated hooks into GitHub source (for instance)
- Uses Buildah to build and push OCI images
  - No need to instrument a running Docker Daemon inside of your build execution
- Staged Pipeline established for sequential build step needs without thousands of plugins
  - The build pipeline simply initiates containers as declaratively defined
- Automatic deployment hooks
  - Knative Services (service.serving.knative.dev) will be deployed that will have specific deployment behaviour

KNATIVE BUILD PIPELINE ANATOMY

What is Buildah?

- At its most simple, **Buildah** is an ***OCI build tool*** used to create, pull, and push images to image registries
- Buildah can create **images from scratch** or add to/change existing images via its tooling idioms with support for all Docker's current functionality plus some more such as buildah-add
- Buildah can commit layers at user determined points in the build process, i.e. not until the end
- Simple fork-exec model and **does not run as a daemon**
- Buildah **Bud** ("build using docker") allows use of existing Dockerfiles

buildah

https://buildah.io/

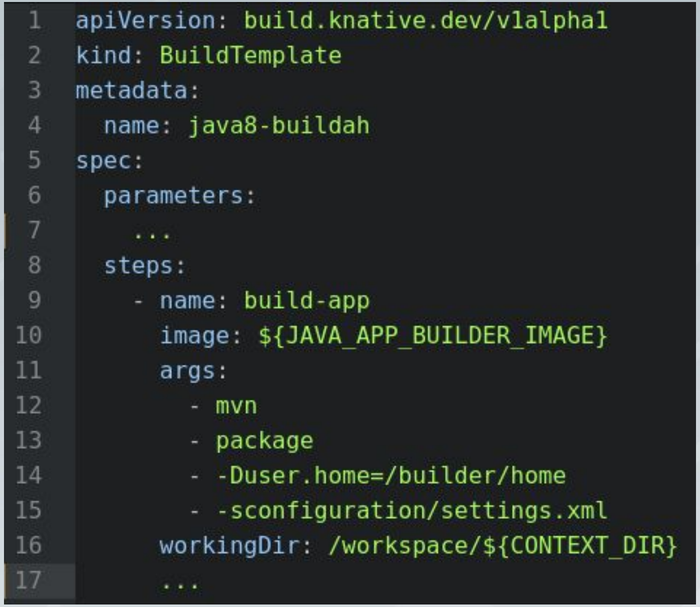
KNATIVE BUILD PIPELINE ANATOMY

`buildtemplate.build.knative.dev/v1alpha1`

Build primitive to initiate a build and:

- Define a Service Account with which to run the build
- Instruments how, where, and what tag to build
- Defines a Build Template to use and pipes in relevant arguments for execution
- Defines a persistent volume claim to cache assets during the build

```
oc apply -f
https://github.com/mike-costello/knative-build-samples/blob/master/build/camel-simple-build.yaml
```



```yaml
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: java8-buildah
spec:
  parameters:
    ...
  steps:
    - name: build-app
      image: ${JAVA_APP_BUILDER_IMAGE}
      args:
        - mvn
        - package
        - -Duser.home=/builder/home
        - -sconfiguration/settings.xml
      workingDir: /workspace/${CONTEXT_DIR}
      ...
```

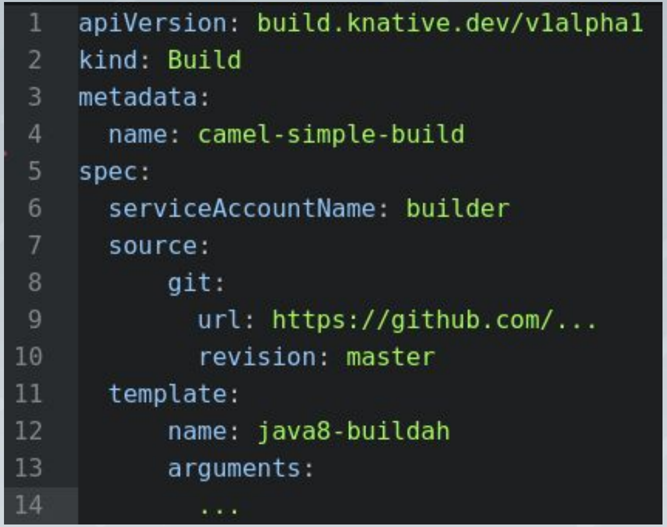
KNATIVE BUILD PIPELINE ANATOMY

build.build.knative.dev/v1alpha1

SImple step based build pipeline that uses:

- The Official Sonatype Maven image to build a jdk8 maven based Java application
- A Buildah image to construct and deploy images for push to a registry (in our case Quay.io)
- A Build revision that is now appropriate to be used by Knative Configuration to feed Knative Serving services that instrument specific revision behaviour

```
oc apply -f
https://github.com/mike-costello/kna
tive-build-samples/blob/master/build
/java8-buildah-template.yaml
```



```yaml
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: camel-simple-build
spec:
  serviceAccountName: builder
  source:
      git:
        url: https://github.com/...
        revision: master
  template:
      name: java8-buildah
      arguments:
        ...
```

STOP! Demo Time!

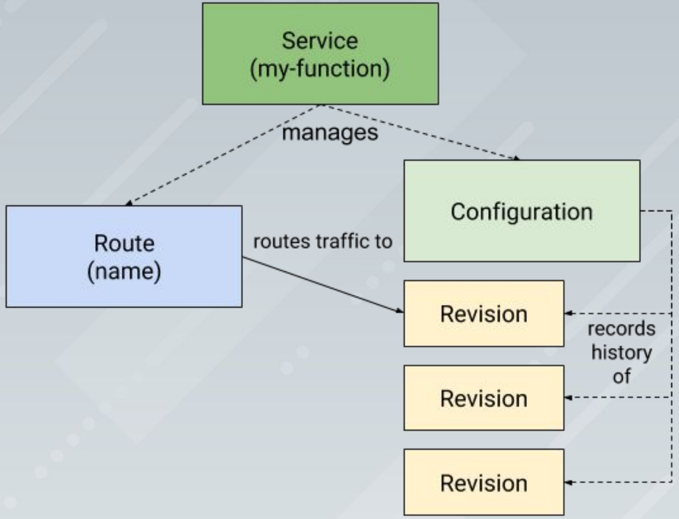
KNATIVE SERVING

Knative Serving 90k Foot View

- A set of **K8s primitives** that provide:
  - **Rapid deployment** of **serverless** containers
  - **Automatic scaling** up and down to zero
  - **Routing and network** programming for Istio components
  - **Point-in-time snapshots** (revisions) of deployed code and configuration

Knative Serving Resources

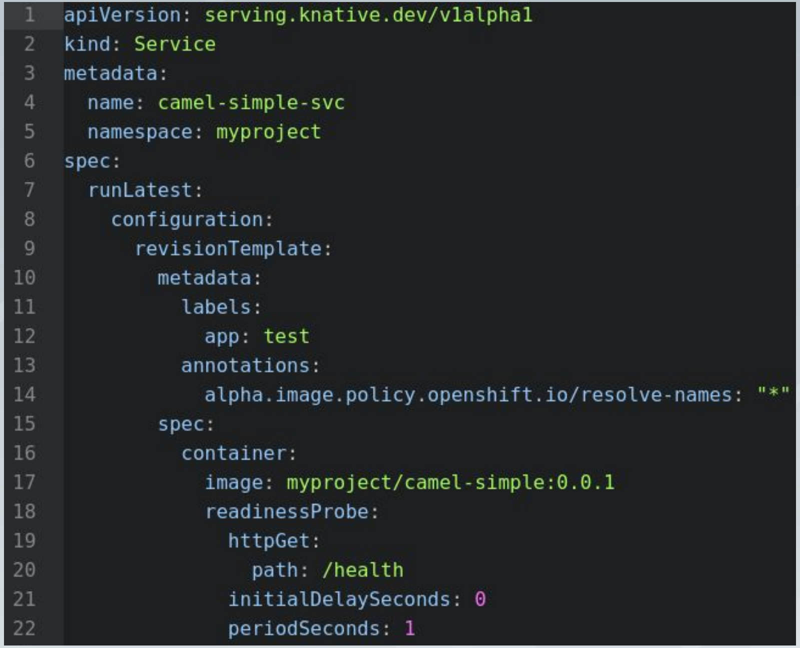
- *Service* (service.serving.knative.dev)
  - Lifecyle management for services
    - Ensures application has a route, a configuration, and a new revision for each update/revision of the service.
  - Routing for service revisions
- *Route* (route.serving.knative.dev)
  - Maps an endpoint to a one or more revisions.
  - Fractional traffic and named routes.
- *Configuration* (configuration.serving.knative.dev)
  - Maintains deployment state.
  - Modifying a configuration creates a new revision.
- *Revision* (revision.serving.knative.dev)
  - Point-in-time snapshot of the code
  - Configuration for each modification
  - Revisions are immutable objects
  - Maintained into perpetuity



Knative Serving

`service.serving.knative.dev/v1alpha1`

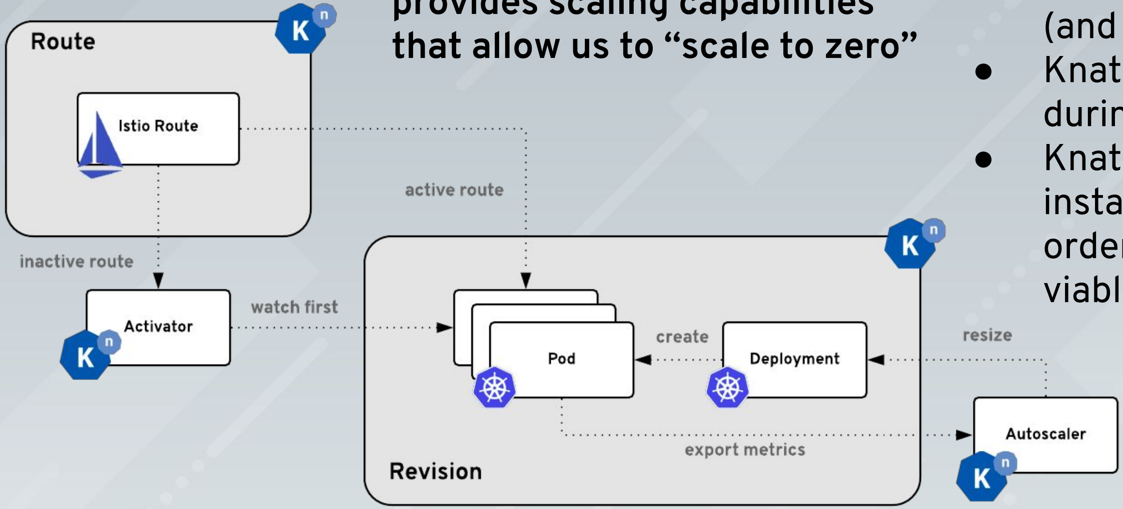
- Knative Services provisioned via image coordinates
- Contains the same pod spec that we know and love
- Abstracts the primitives (building blocks) that we're used to such as K8 services, deployments, etc.



```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: camel-simple-svc
  namespace: myproject
spec:
  runLatest:
    configuration:
      revisionTemplate:
        metadata:
          labels:
            app: test
          annotations:
            alpha.image.policy.openshift.io/resolve-names: "*"
        spec:
          container:
            image: myproject/camel-simple:0.0.1
            readinessProbe:
              httpGet:
                path: /health
              initialDelaySeconds: 0
              periodSeconds: 1
```

SCALE TO ZERO- and beyond!

Knative Serving - *(service.serving.knative.dev)* provides scaling capabilities that allow us to "scale to zero"



- Autoscaling is not new to OpenShift, but we lack the facilities to scale to zero (and cold start from zero)
- Knative will spin all containers down during quiet periods (no traffic)
- Knative will spin up a new service instance when it is initially created in order to validate that the revision is viable (passing health checks)

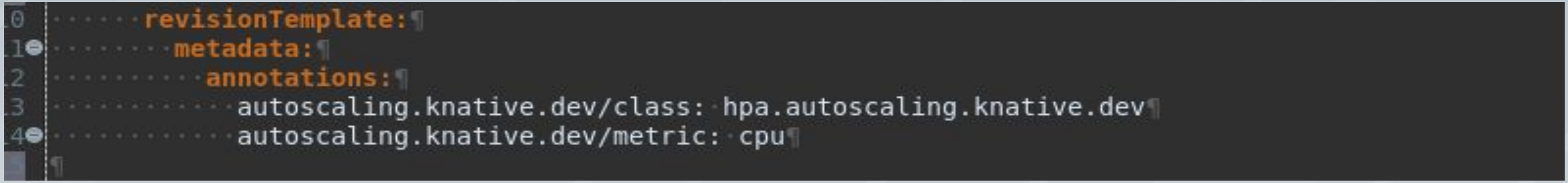
AutoScaling - 0 and beyond!

- Knative Serving (service.serving.knative.dev) provides **scaling** that scales to the **number of inflight concurrent** requests
    - **Default scaling** at 100 inflight concurrent requests
        - Takes about a minute to stabilize to the desired level of concurrency as algorithmically we sample inflight concurrent requests over a 60 second period
        - Number of inflight concurrent requests to trigger scaling is configurable

    - ***"Panic at the Disco"***
        - A 6 second "panic" window gets entered if we exceed 2x targeted concurrent requests

AutoScaling 0 and beyond! (*Continued*)

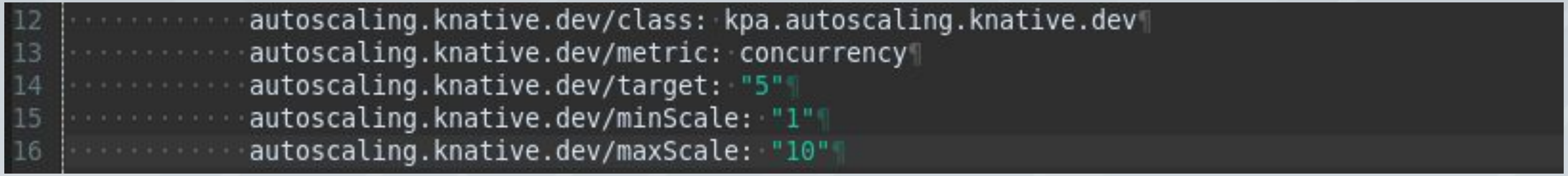
- ***"Panic at the Disco" (Continued)***
  - AutoScalers can be configured to move beyond the idea of concurrency and can evaluate other things like CPU usage



```yaml
      revisionTemplate:
        metadata:
          annotations:
            autoscaling.knative.dev/class: hpa.autoscaling.knative.dev
            autoscaling.knative.dev/metric: cpu

```

- Or of course, we can simply use **concurrent requests**



```yaml
            autoscaling.knative.dev/class: kpa.autoscaling.knative.dev
            autoscaling.knative.dev/metric: concurrency
            autoscaling.knative.dev/target: "5"
            autoscaling.knative.dev/minScale: "1"
            autoscaling.knative.dev/maxScale: "10"
```

  - As we hit **"panic"** stages, we attempt to stabilize over a 60 second window and if we can, we'll return back to our "normalized" stable concurrency scale/state

STOP! Demo Time!

KNATIVE EVENTING

(this specification is **in-flux** but for now we understand the following things)

Knative Eventing: The 90K Foot View

- ***Services are loosely coupled.*** These services can be developed and deployed independently on, and across a variety of platforms (for example Kubernetes, VMs, SaaS or FaaS).
- ***Event producers and event consumers are independent.*** Any producer (or source), can generate events before there are active event consumers that are listening. Any event consumer can express interest in an event or class of events, before there are producers that are creating those events.
- Other services can be connected to the Eventing system. These services can perform the following functions:
  - Create ***new applications without modifying the event producer or event consumer.***
  - Select and target specific subsets of the events from their producers.
- Cross-service interoperability.
  - CloudEvents specification that is developed by the CNCF Serverless WG.

Knative Event Consumers

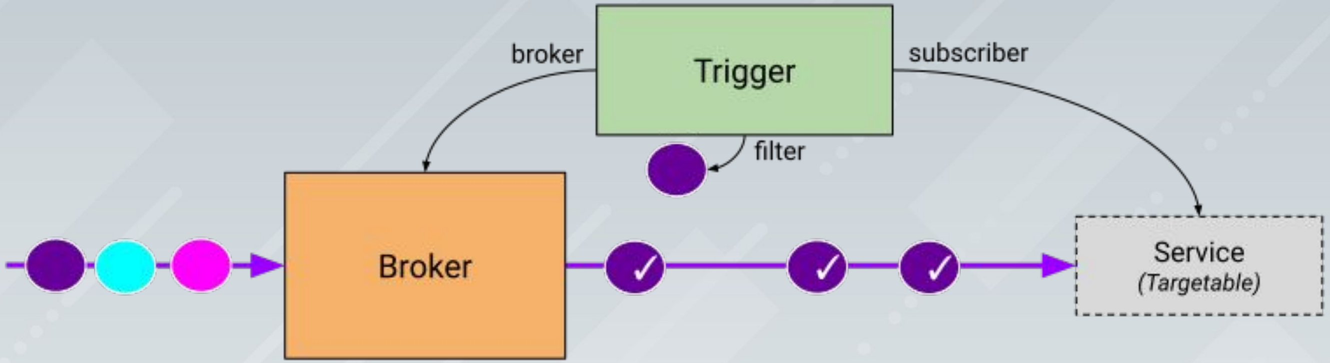
Knative Eventing defines two generic interfaces that can be implemented by multiple Kubernetes resources:

1. **Addressable** objects are able to receive and acknowledge an event delivered over HTTP to an address defined in their `status.address.hostname`field. As a special case, the core [Kubernetes Service object]() also fulfils the Addressable interface.
2. **Callable** objects are able to receive an event delivered over HTTP and transform the event, returning 0 or 1 new events in the HTTP response. These returned events may be further processed in the same way that events from an external event source are processed.

Event Brokers And Triggers (90K Ft. View)

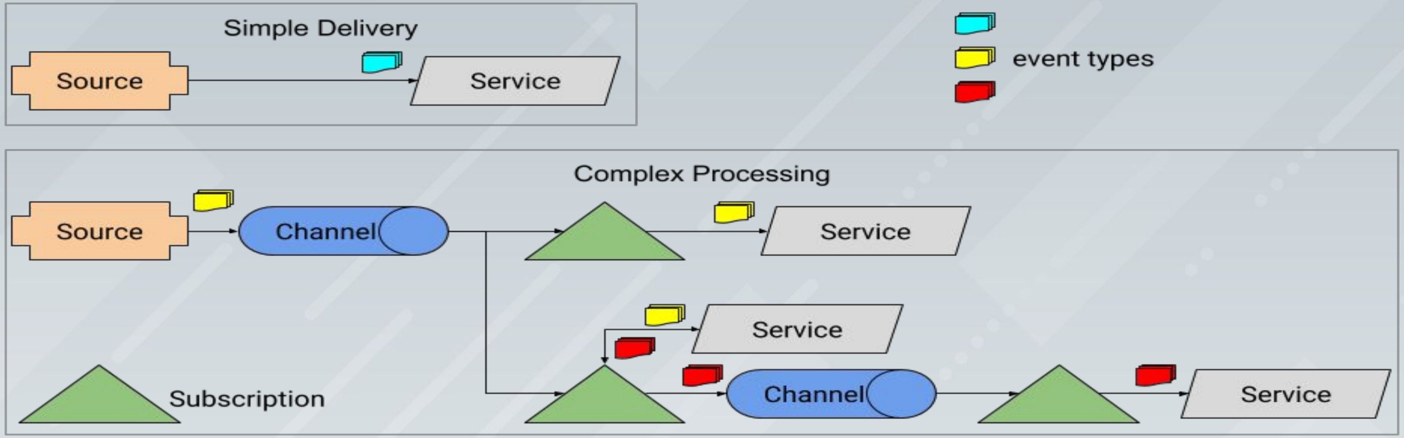
(new in v0.5)

- As of v0.5, Knative Eventing defines **Broker** and **Trigger** objects to make it easier to filter events.
- A Broker provides a **bucket of events** which can be selected by attribute. It receives events and forwards them to **subscribers** defined by one or more matching **Triggers.**
- A **Trigger** describes a filter on event attributes which should be delivered to an **Addressable.** You can create as many Triggers as necessary.



Knative Eventing Pub/Sub & Channels

- **Channel**. Messaging implementations may provide implementations of Channels via the ClusterChannelProvisioner object.
- Events are delivered to Services or forwarded to other channels (possibly of a different type) using Subscriptions.
- Message delivery in a cluster to vary based on requirements, so that some events might be handled by an in-memory implementation while others would be persisted using Apache Kafka or NATS Streaming.

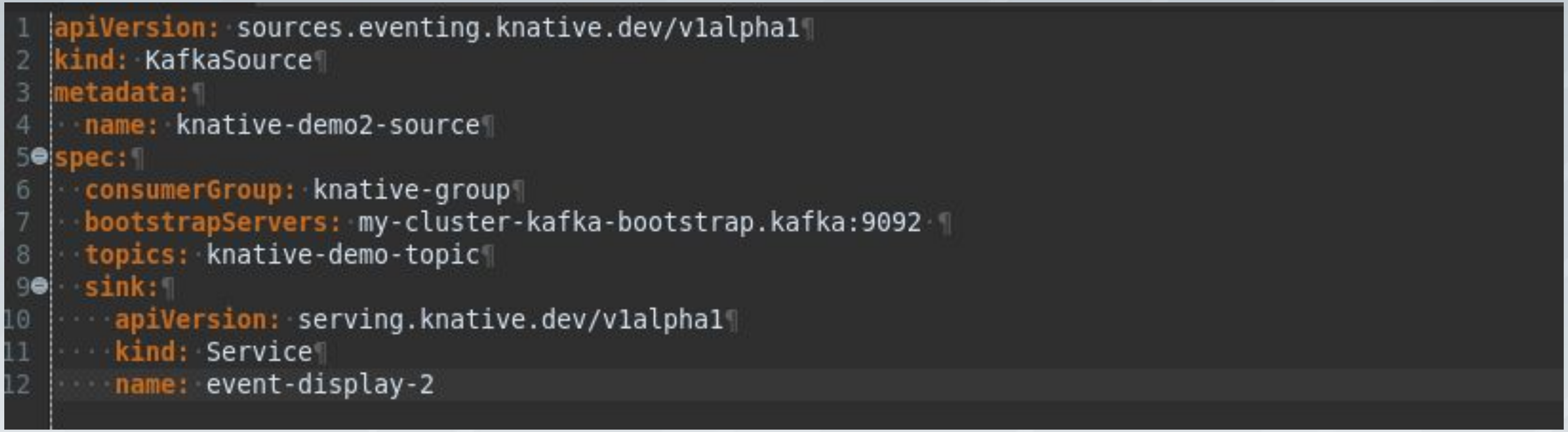


Knative Event Sources
(And our Kafka Source)

- At a high level, ***Event Sources*** are K8s Custom Resources which provide a mechanism for registering interest in a class of events from a particular software system
- There are several implementations out there in the wild
  - K8s Events
  - Kafka
  - Apache Camel (great stuff happening here. Please check out Camel K)
  - GCP PubSub
  - Github
  - More to come...

Knative Event Sources
(And our Kafka Source)

- A Sample Kafka Source



```yaml
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: KafkaSource
metadata:
  name: knative-demo2-source
spec:
  consumerGroup: knative-group
  bootstrapServers: my-cluster-kafka-bootstrap.kafka:9092
  topics: knative-demo-topic
  sink:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: event-display-2
```

Way too Quick and A Bunch of other Stuff we don’t have time for REALLY needs to be addressed....but…

STOP! Demo Time!

RED HAT SUMMIT

THANK YOU

linkedin.com/company/Red-Hat

facebook.com/RedHatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat