Kubernetes Patterns

Patterns, Principles, and Practices for Designing Cloud Native Applications

Bilgin Ibryam & Roland Huss

Kubernetes Patterns

Patterns, Principles, and Practices for Designing Cloud Native Applications

Bilgin Ibryam and Roland Huß

This book is for sale at http://leanpub.com/k8spatterns

This version was published on 2017-07-12

Tweet This Book!

Please help Bilgin Ibryam and Roland Huß by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Kubernetes Patterns by @bibryam and @ro14nd #k8sPatterns

The suggested hashtag for this book is #k8sPatterns.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#k8sPatterns

To our families

Contents

Introduction

With the evolution of Microservices and containers in the recent years, the way we design, develop and run software has changed significantly. The new modern applications are optimised for scalability, elasticity, failure, and speed of change. Driven by new principles, these modern architectures require a different set of patterns and practices to be applied. In this book, we will try to cover all these new concepts in breadth rather than depth. But first, let's have a brief look at the two major ingredients of this book: Kubernetes and Design Patterns

Kubernetes

Kubernetes is a container orchestration platform. The origin of Kubernetes lies somewhere in the Google data centres where it all started from Google's internal platform Borg. Google uses Borg since many years for orchestrating applications which run in an custom container format. In 2014 Google decided to open source Borg as "Kubernetes" (Greek for "helmsman" or "pilot") and in 2015 Kubernetes was the first project donated to the newly founded Cloud Native Computing Foundation (CNCF).

Right from the start Kubernetes immediately took off and raised a lot of interest. A whole community of users and contributor grew around Kubernetes at incredibly fast pace. Today, Kubernetes is measured as one of the most active projects on GitHub. It is probably fair to claim that currently (2017) Kubernetes is the most used and feature rich container orchestration platform. Kubernetes also forms the foundation of other platforms which are built on top of it. The most prominent of those Platform-as-a-Service systems is OpenShift which provides various additional capabilities to Kubernetes, including ways to build applications within the platform. These are only some of the reasons why we chose Kubernetes as the reference platform for the cloud native patterns in this book.

Design Patterns

The idea of Design Patterns dates back until the 1970s. Originally this idea was not developed within the field of IT but comes from architecture. Christopher Alexander, an architect and system theorist, and his team published the groundbreaking "A Pattern Language" in 1979 where they describe architectural patterns for creating towns, buildings and other constructions. Sometime later this idea swapped over to the just born software industry. The most famous book in this area is "Design Patterns - Elements of Reusable Object-Oriented Software" by the Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides ("The gang of four"). When we talk about "Singletons", "Factories", or "Delegation", it's because of this defining work. Many other great pattern books have been written since that for various fields at different levels of granularity like Enterprise Integration Patterns or Camel Design Patterns.

The essential nature of a Pattern is that it describes a *repeatable solution to a problem*[1]. It is different to a recipe as it supposed to solve many similar problems in a similar way instead of providing detailed step-by-step instructions. For example, the Alexandrian pattern "Beer House (90)" describes how public drinking houses should be constructed where "strangers and friends are drinking companions" and not "anchors of the lonely". All pubs built after this patterns will look differently, but they share common characteristics like open alcoves for groups of four to eight, and a half-dozen activities so that people continuously move from one to another.

However, it's not only about the solution. As the name of the original book implies, patterns can also form a *language*. It's a dense, noun centric language in which each pattern carries a unique *name*. The purpose of the name is, that when the language is established, everybody "speaking" this language builds up an instant similar mental representation of this pattern. The situation is much like when we talk about a table, anyone speaking English automatically associates with it a thing of it which has four legs and a top where you can put things on. The same happens for us software engineers when we speak about a "Factory". In an object oriented programming

[1] The original definition in the context of architecture was defined by Christopher Alexander et. al. as that "*each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*" ("A Pattern Language", Christopher Alexander et. al). We think, that when transformed this into our field, that it pretty good matches the patterns which we describe in this book except that we probably have not that significant variation in the patterns' solution.

language context, we immediately associate an object which produces other objects. That way we can easily leverage the communication on a higher level and move on with the real interesting, yet unsolved problems.

There are other characteristics of a pattern language. Patterns are interconnected and can overlap so that they build a graph that covers (hopefully) the full problem space. All patterns can be applied in a certain context, which is part of the pattern descriptions. Also, as already laid out in the original "A Pattern Language", patterns need not have the same granularity and scope. More general patterns cover a wider problem space and provide a rough guidance how to solve the problem. Finer granular patterns have a very concrete solution proposal but are not as widely applicable. This book contains all sort of patterns, where many patterns reference other patterns or might even include other patterns as part of the solution.

Another feature of patterns is that they follow a fixed format. There is no standard form how patterns are described. There have been various variations of the original Alexandrian form with different degrees of granularity. An excellent overview of the formats used for pattern languages is given in Martin Fowler's Writing Software Patterns. For the purpose of this book, we choose a pragmatic approach for our patterns. It is worth setting your expectations right from the very beginning by saying that this book does not follow any particular pattern description language. For each pattern we have chosen the following structure:

- **Name**: the pattern name;
- **Logline**: a short description and purpose of the pattern. This description (in *italics*) comes right after the name;
- **Problem**: when and how the problem manifests itself;
- **Solution**: how the pattern solves the problem in a Kubernetes-specific way;
- **Discussion**: the pros & cons of the solution and when to apply it;
- **More Information**: other information sources related to the topic.

Each Pattern has a *name* which is in the centre of our small pattern language. The logline captures the essence of each pattern. The *logline* is provided as one or two sentences in italics right after the name. The *problem* section then gives the wider context and describes the pattern space in details before the *solution* section explains the particular solution for this problem. These sections also contain cross

references to each other patterns which are either related or even part of the given pattern. A *discussion* also refers to the pattern's context and discusses the advantages and disadvantages of this particular pattern. In the final section, we give further references to additional content which might be helpful in the context of this pattern.

The book has been written in a relaxed style; it is similar to a series of essays, but with this consistent structure.

Who this Book is for

There are plenty of good Kubernetes books covering how Kubernetes works at varying degrees. This book is intended for developers who are somewhat familiar with Kubernetes concepts but are perhaps lacking the real world experience. It is based on use cases, and lessons learnt from real-world projects with the intention of making the reader think and become inspired to create even better cloud native applications.

Downloading the Source Code

The source code for the examples in this book is available online from github at https://github.com/k8spatterns

About the Authors

Bilgin Ibryam is a software craftsman based in London, a senior integration architect at Red Hat, and a committer for Apache Camel and Apache OFBiz projects. He is an open source fanatic, and is passionate about distributed systems, messaging, and enterprise integration patterns, and application integration in general. His expertise includes Apache Camel, Apache ActiveMQ, Apache Karaf, Apache CXF, JBoss Fuse, JBoss Infinispan, etc. In his spare time, he enjoys contributing to open source projects and blogging. Follow Bilgin using any of the channels below:

Twitter: https://twitter.com/bibryam
Github: https://github.com/bibryam
Blog: http://www.ofbizian.com

Dr. Roland Huss is a software engineer at Red Hat working in the JBoss Fuse team to bring EAI to OpenShift and Kubernetes. He has been developing in Java for twenty years now but never forgot his roots as system administrator. Roland is an active open source contributor, lead developer of the JMX-HTTP bridge Jolokia and the popular fabric8io/docker-maven-plugin. Beside coding he likes to spread the word on conferences and writing. And yes, he loves growing chilli pepper.

Twitter: https://twitter.com/ro14nd
Github: https://github.com/rhuss
Blog: https://ro14nd.de

Technical Reviewers

Paolo Antinori is an ex-consultant and a current Software Engineer at Red Hat, working primarily on the JBoss Fuse product family. He likes to challenge himself with technology-related problems at any level of the software stack, from the operating system to the user interface. He finds particular joy when he invents "hacks" to solve or bypass problems that were not supposed to be solved.

Andrea Tarocchi is an open source and GNU/Linux enthusiast who now works at Red Hat. Computer-passionate since he was eight years old, he has studied Computer Engineering in Florence and has worked mainly in the application integration domain. If you would like to put a face to the name, check out his https://about.me/andrea.tarocchi profile.

Proofreader

I Foundational Patterns

A pattern does not always fit into one category. Depending on the intent, the same pattern may have multiple implications and contribute to multiple categories. The pattern categories in this book are loosely defined, mainly to provide structure for the book.

The patterns in this category are used for creating containerized applications. These are fundamental patterns and principles that any any application regardless if its nature has to implement in order to become a good cloud native citizen.

1. Automatable Unit

Kubernetes offers new set of distributed primitives and encapsulation techniques for creating automatable applications.

Problem

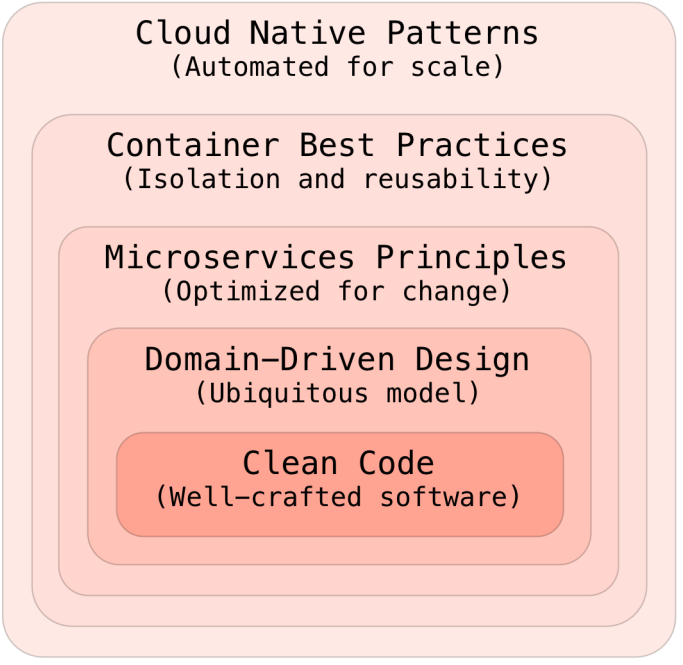
The Microservices architectural style tackles the business complexity of software through modularization of business capabilities. And as part of the Microservices movement there is a great amount of theory and supplemental techniques for creating Microservices from scratch or splitting monoliths into Microservices. Most of that theory and practices are based on Domain-Driven Design book from Eric Evans and its concepts of Bounded Contexts and Aggregates. Bounded Contexts deal with large models by dividing them into different components and Aggregates help further to group Bounded Contexts into modules with defined transaction boundaries. But in addition to these business domain conciderations, for every Microservice, there are also technical concerns around its organization and structure in order to fit and benifit from the platform it is running on. Containers and container orchestrators such as Kubernetes provide many primitives and abstractions to address the different concerns of distributes applications and here we are discussing various options to consider and design directions to make.

Solution

In this chapter there is not a pattern that we analyse. Instead we set the scene for the rest of the book and explain few of the Kubernetes concepts, good practices and principles to keep in mind, and how they relate to the cloud native application design.

The Path to Cloud Native

Throughout this book we look at container interactions by treating the containers as black boxes. But we created this section just to emphasis the importance of what goes into containers. Containers and cloud native platforms will bring tremendous benefits to your application and team, but if all you put in containers is rubbish, you will get rubbish applications at scale.



What you need to know to create good cloud native applications

To give you some perspective on the various levels of complexity that require different set of skills:

- At the lowest level, every variable you define, every method you create, every class you decide to instantiate will play a role in the long term maintenance burden of your application. And no mather what container technology and orchestration platform you use, the biggest impact on your application will be

done by the development team and the artifacts they create. So it is important to have developers who strive to write clean code, have good amount of automated tests, refactor constantly to improve code quality, and generally are software craftsman by heart in the first place.

- Domain-Driven Design is about approaching software design from a business perspective with the intention of keeping the design as much close to the real world as possible. This works best for Object-oriented programing languages, but there are also other good ways for modeling and designing software for real world problems. The reason I gave it here as an example is that, on top of the clean code, which is more about technicalities of programing, there is this next level of abstraction where you have to analyse and express the business problem in the application. So having a model with the right business and transaction boundaries, easy to consume interfaces and beautiful APIs is the foundation for successful containarization and automation.
- Microservices architectural style was born as a variation and alternative to SOA to address the need of fast release cycles. And it very quickly evolved to be the norm and provided valuable principles and practices for designing distributed applications. Applying these principles will let you create applications that are optimized for scale, resiliency and pace of change which are common requirements for any modern software solution build today.
- Containers have been also very quickly adopted as the standard way of packaging and running distributed applications. And even if it is a more of a technicality, there are many container patterns and best practices to follow. Creating modular, reusable containers that are good cloud native citizens is another fundamental prerequisite.
- With growing number of Microservices in every organization, comes the need to manage them using more effective methods and tools. Cloud native is a relatively new term used to describe principles, patterns, and tools used to automate containerized Microservices at scale. I use cloud native interchangeably with Kubernetes, which is the most popular cloud native platform implementation available today. In this book, we will be mainly exploring patterns and practices addressing the concerns of this level, but for any of that to work as expected, we need all the previous levels implemented properly as well.

Containers

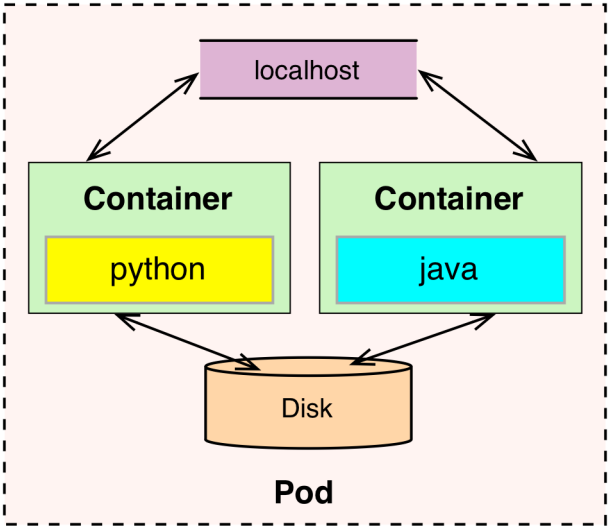
Containers are the building blocks for Kubernetes based cloud native applications. If we have to make a parallel with OOP and Java, container images are like classes, and container instances are like the objects. The same way we can extend classes, reuse and alter behaviour, we can have container images that extend other container images to reuse and alter behaviour. The same way we can do object composition and use functionality, we can do container compositions by putting containers into a pod and have collaborating container instances. If we continue the parallel, Kubernetes would be like the JVM but spread over multiple hosts, and responsible for running and managing the container instances. Init containers would be something like object constructors, daemon containers would be similar to daemon threads that run in the background (like the Java Garbage Collector for example). A pod would be something similar to a Spring Context where multiple running objects share a managed lifecycle and can access each other directly. The parallel doesn't go much further, but the point is that containers play a fundamental role in Kubernetes and creating modularized, reusable, single purpose containers is fundamental for the long term success of any project and even the containers ecosystem as a whole. Below are few links that describes good container practices to follow. Apart from the technical characteristics of a container which provide packaging and isolation, what does a container represent and what is its purpose in the context of a distributed application? Here are few to pick from.

- A container is the boundary of a unit of functionality that addresses a single concern.
- A container is owned by one team, and has its own release cycle.
- A container is self contained, defines and carries its own build time dependencies.
- A container is immutable and don't change once it is build, but only configured.
- A container has defined runtime dependencies and resource requirements.
- A container has well defined APIs to expose its functionality.
- A container instance runs as a single well behaved unix process.
- A container instance is disposable and safe to scale up or down at any moment.

In addition to all these characteristics, a good container is modular. It is parameterised and created for reuse. It is paremetrized for the different environments it is going to run, but also parameterised for the different use cases it may be part of. Having small, modular reusable containers leads to creation of more specialized and stable containers in the long term, similarly to a good reusable library in the programing languages world.

Pods

If we look at the characteristics of containers, they are a perfect match for implementing the Microservices principles. A container provides single unit of functionality, belongs to a single team, has independent release cycle, provides deployment and runtime isolation. Seems like one Microservice corresponds to one container. And that is true for most of the time. But most cloud native platforms offer another primitive for managing the lifecycle of a group of containers. That in Kubernetes is called a pod, in AWS ECS and Apache Mesos it is called Task Group, etc. A pod is an atomic unit of scheduling, deployment and runtime isolation for a group of containers. All containers in a pod end up scheduled always to the same host, deployed together whether that is for scaling or host migration purposes and also can share file system and process namespaces (PID, Network, IPC). This allows for the containers in a pod to interact with each other over file system, localhost or host IPC mechanisms if desired (for performance reasons for example).



Pod as the runtime container management unit

As you can see, at development and build time, a Microservice corresponds to a container that one team develops and releases. But at runtime a Microservice corresponds to a pod which is the unit of deployment, placement and scaling. And sometimes, a pod may contain more than one containers. One such an example would be when a containerized Microservice uses a helper container at runtime as the SideCar pattern chapter demonstrates later. Containers and pods, and their unique characteristics offer a new set of patterns and principles for designing Microservices based application. We have seen some of the characteristics of good containers previously, now let's see the characteristics of a good pod.

- A pod is atomic unit of scheduling. That means the scheduler will try to find a host that satisfy the requirements of all containers that belongs to the pod together (there are some specifics around init-containers which form another sub-group of containers with independent lifecycle and resource requirements). If you create a pod with many containers, the scheduler needs to find a host that have enough resources to satisfy all container demands combined.
- A pod ensures co-location of containers. Thanks to the collocation, containers in the same pod have additional means to interact with each other. The most

common ways for communication includes using a shared local file system for exchanging data, or using the localhost network interface, or some kind of host IPC (Inter-Process Communication) mechanism for high performance interactions.

- A pod is the unit of deployment for one or more containers. The only way to run a container whether that is for scale, migration, etc is through the pod abstraction.
- A pod has its own IP address, name and port range that is shared by all containers belonging to it. That means containers in the same pod have to be carefully configured to avoid port clashes.

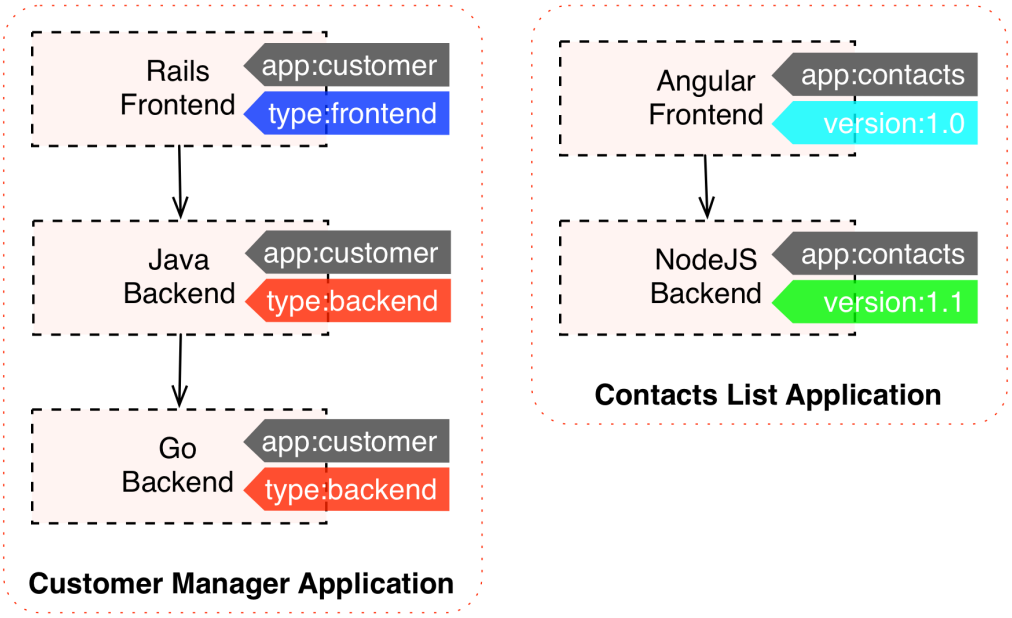
Services

Pods are ephemeral, they can come (during scaling up) and go (during scaling down) at any time. A pod IP is known only after it is scheduled and started on a node. A pod can be re-scheduled to a different node if the existing node it is running on is not healthy any longer. All that means the pod IP may change over the life of an application, and there is a need for another primitive for discovery and load balancing. This is where the Kubernetes services come into play. The service is another simple but powerful Kubernetes abstraction that binds the service name to an IP address and port number in a permanent way. So a service represents a named entry point for a piece of functionality provided somewhere else. In the most common scenario, the service will represent the entry point for a set of pods, but that might not be always the case. The service is a generic primitive and it may also point to functionality that is provided outside of the Kubernetes cluster. As such, the service primitive can be used for service discovery and load balancing, and allows altering implementations and scaling without affecting service consumers.

Labels

We have seen that a Microservice is a container at build time, but represented by a pod at runtime. That raises the question what is an application that consist of multiple Microservices? And here Kubernetes offers two more primitives that can help you define the concept of application - that is labels and namespaces. In the past, an application corresponded to a single deployment unit, with a single versioning

scheme and release cycle. There was a direct physical representation of the concept of application in the form of a .war, or .ear or some other packaging format. But once an application is split into independent Microservices that are independently developed, released, run, restarted, scaled, etc. the importance of the concept of application diminishes. There are not any longer important activities that have to performed at application level, instead they are perform at a Microservice level. But if you still need a way to indicate that a number of independent services belong to an application, labels can be used for this purpose. Let's imagine that we have split one monolithic application into 3 Microservices, and another application into 2 Microservices. We have now 5 pods definitions (and may be many more pod instances) that are totally independent from development and runtime point of view. But we may still need to indicate that the first 3 pods represent an application and the other 2 pods represent another application. Even the pods are independent, in order to provide a business value they may depend from one another. For example one pod may contain the containers responsible for the frontend, and the other two pods responsible to provide the backend functionality. Having either of these pods down, will make the whole application useless from business point of view. Using labels allows us to group and manage multiple pods as one logical unit.



Labels as an application identity

Here are few examples where the labels can be useful:

- Labels are used by the replication controller to keep a number of instances of a specific pod running. That means every pod definition needs to have unique combination of labels used for scheduling.
- Labels are also used heavily by the scheduler. In order to place pods on the hosts that satisfy the pods' requirements, the scheduler uses labels for co-locating or spreading pods.
- A label can indicate whether a pod belongs to a logic group of pods (such as application) and only together this group of pods can provide a business value. A label can give application identity to a group of pods and containers.
- In addition to these common use cases, labels can be used to track any kind of meta data describing a pod. It might be difficult to guess what a label might be useful for in the future, so it is better to have enough labels that describe any important aspect of the pod to begin with. For example, having labels to indicate the logical group of an application, the business characteristics

and criticality, the specific runtime platform dependencies (such as hardware architecture, location preferences), etc are all useful. Later these labels can be used by the scheduler for more fine grained scheduling, or the same labels can be used from the command line tools for managing the matching pods at scale.

Annotations

There is another primitive that is very similar to labels called *annotations*. Like labels, annotations are key-value maps, but they are intended for specifying non-identifying metadata and for machine usage rather than human. The information on the annotations is not intended for querying and matching objects, instead it is intended for attaching additional metadata to objects that can be used from various tools and libraries. Some examples for using annotations includes build Ids, release Ids, image information, timestamps, git branch names, PR numbers, image hashes, registry addresses, author names, tooling information, etc. So while labels are used primarily for query matching and then performing some actions on the matching resources, annotations are used to attach machine consumable metadata.

Namespaces

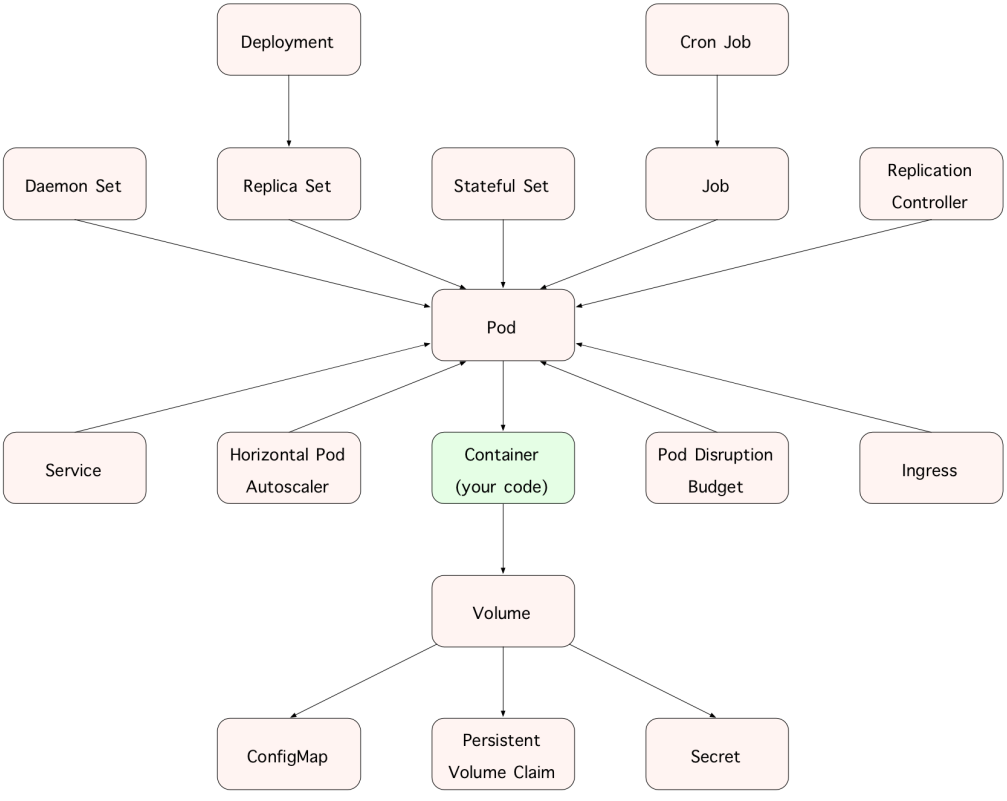
Labels allow tagging of various Kubernetes resources such as pods. Then these labels can be used for managing groups of pods (or other resources) for different purposes such as scheduling, deployment, scaling, migration, etc. Another primitive that can also help for the management of a group of resources is the Kubernetes namespace. As it is described, a namespace may seem similar to a label, but in reality it is a very different primitive with different characteristics and purpose. Kubernetes namespaces allow isolating a Kubernetes cluster (which is usually spread across multiple hosts) into logical pool of resources. Namespaces provide scopes for Kubernetes resources and a mechanism to apply authorizations and other policies to a subsection of the cluster. The most common example of namespaces is using them to represent the different software environments such as *development*, *testing*, *integration testing* or *production*. In addition, namespaces can also be used to achieve multi-tenancy, provide isolation for team workspaces, projects and even specific applications. But ultimately, namespace isolation is no better than having separate cluster isolation which seems to be the more common setup. Typically, there is one

non-production Kubernetes cluster which is used for a number of environments (*development*, *testing*, *integration testing*) and another production Kubernetes cluster to represent *performance testing* and *production* environments. Let's see some of the characteristics of namespaces and how namespaces can help us in different scenarios:

- A namespace provides a scope for resources such as containers, pods, services, replication controllers, etc. Names of resources need to be unique within a namespace, but not across namespaces.
- By default namespaces provide a scope for resources, but there is nothing isolating those resources and preventing access from one resource to another. For example a pod from dev namespace can access another pod from prod namespace as long as the pod IP address is known. But there are Kubernetes plugins which provide networking isolation to achieve true multi-tenancy across namespaces if desired.
- Some other resources such as namespaces themselves, nodes, persistent volumes, do not belong to namespaces and should have unique cluster wide names.
- Each Kubernetes service belongs to a namespace and gets a corresponding DNS address that has the namespace in the form of `<service-name>.<namespace-name>.svc.cluster.local`. So the namespace name will be in the URI of every service belonging to the given namespace. That's why it is important to name namespaces wisely.
- Resource Quotas provides constraints that limit the aggregated resource consumption per namespace. With Resource Quotas, a cluster administrator can control the quantity of objects per type that can be created in a namespace (for example a developer namespace may allow only 5 ConfigMaps, 5 Secrets, 5 Services, 5 replicationcontrollers, 5 persistentvolumeclaims and 10 pods).
- Resource Quotas can also limit the total sum of compute resources that can be requested in a given namespace (for example in a cluster with a capacity of 32 GiB RAM and 16 cores, it is possible to allocate half of the resources - 16 GB RAM and 8 cores for the Prod namespace, 8 GB RAM and 4 Cores for PreProd/Perf environment, 4 GB RAM and 2 Cores for Dev, and the same amount for Test namespaces). This aspect of namespaces for constraining various resources consumptions is one of the most valuable ones.

Discussion

In this introductionary chapter we covered only the main Kubernetes concepts that will be used in the rest of the book. But there are more primitives used by developers on day-by-day basis. To give you an idea, if you create a containerized application, here are the collections of Kubernetes objects which you will have to choose from in order to benefit from Kubernetes to a full extend.



Kubernetes concepts for developers

Keep in mind these are only the objects that are used by developers to integrate a containerized application into Kubernetes. There are also other concepts primarily used by Admin/Ops team members to enable developers on the platform and manage

the platform effectively.

More Information

Container Best Practices by Project Atomic
Best practices for writing Dockerfiles by Docker
Container Patterns by Matthias Lubken General Docker Guidelines by OpenShift
Pods by Kubernetes Labels and Selectors by Kubernetes
Annotations by Kubernetes
Namespaces by Kubernetes
Resource Quotas by Kubernetes

2. Predictable Demands

The foundation of a successful application placement and co-existence on a shared cloud environment is based on identifying and declaring the application resource requirements.

Problem

Kubernetes can manage applications written in different programming languages as long as the application can be run in a container. But different languages have different resource requirements. For example any compiled language will run faster but at the same time it will require more memory compared to Just-in-Time runtimes, or interpreted languages. Considering that many modern programing languages in the same category have pretty similar resource requirements, rather than the language a more important aspect is the domain and the business logic of the application. A financial application will be more CPU intensive than a proxy service, and a batch job for copying large files will be more memory intensive than a simple event processor usually. But even more important than the business logic are the actual implementation details. A file copying service can be made light on memory by making it stream data, or a different application may require a lot of memory to cache data. It is difficult to predict how much resources a container may need to function optimally, and it is the developer who knows what are the resource expectations of a service implementation. Some services will need persistent storage to store data, some legacy services will need a fixed port number on the host system to work properly. Some services will have a fixed CPU and memory consumption profile, and some will be spiky. Defining all these application characteristics and passing it to the managing platform is a fundamental prerequisite for cloud native applications.

Solution

Knowing the resource requirements for a container is important mainly for two reasons. First, with all the runtime dependencies defined and resource demands envisaged, Kubernetes can make intelligent decisions for placing a container on the cluster for most efficient hardware utilization. In an environment with shared resources among large number of processes with different priorities, the only way for a successful co-existence is by knowing the demands of every process in advance. But intelligent placement is only one side of the coin. The second reason why container resource profiles are important is the capacity planning. Based on the individual service demands and total number of services, we can come do some capacity planning for the different environments and come up with the most cost effective host profiles to satisfy the total cluster demand. Service resource profiles and capacity planing go hand-to-hand for a successful cluster management in the long term. Next we will see some of the areas to consider and plan for Kubernetes based applications.

Container Runtime Dependencies

Container file systems are ephemeral and lost when a container is shut down. Kubernetes offers Volume as a pod level storage utility that survives container restarts. The simplest type of Volume is emptyDir which lives as long as the pod lives and when the pod is removed its content is also lost. To have a volume that survives pod restarts, the volume needs to be backed by some other kind of storage mechanism. If your application needs to read or write files to such a long lived storage, you have to declare that dependency explicitly in the container definition using Volumes.

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

The kind of Volume a pod requires will be evaluated by the scheduler and will affect where the pod gets placed. If the pod requires a Volume that is not provided by any node on the cluster, the pod will not be scheduled at all. Volumes are an example of a runtime dependency that affects on what kind of infrastructure a pod can run and whether it can be scheduled at all or not.

A similar dependency happens when you ask Kubernetes to expose a container port on a specific port on the host system through HostPort. Using this approach creates another runtime dependency on the nodes and limits the nodes where a pod can be scheduled. Due to port conflicts, you can scale to as many pods as there are nodes in the Kubernetes cluster.

A different type of dependency are configurations. Almost every application needs some kind of configuration information and the recommended solution offered by Kubernetes is through ConfigMaps. Your services needs to have a strategy for consuming configurations, that is either through environment variables or through the file system. In either case, this introduces a runtime dependency of your container to the named ConfigMaps. If all of the expected ConfigMaps are not created on a namespace, the containers will be placed on a node, but will not start up.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

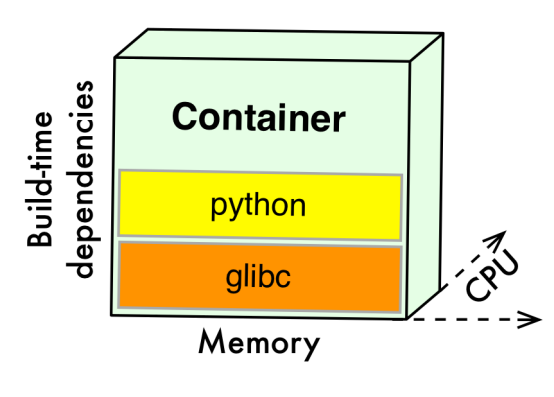
A similar concept to ConfigMap is Secret which offer a slightly more secure way of distributing environment specific configurations to a container. The way to consume a Secret is the same as the ConfigMap consumption and it introduces a similar dependency from a container to a namespace.

While ConfigMap and Secret are pure admin tasks that have to be performed, storage and port numbers are provided by hosts. Some of these dependencies will limit where (if anywhere at all) a pod gets scheduled, and other dependencies may prevent the pod from starting up. When requesting such resources for your pods, always consider the runtime dependencies it will create later.

Container Resource Profiles

Specifying container dependencies such as ConfigMap, Secret and Volumes is straightforward. Some more thinking and experimentation is required for figuring

out the resources requirements of a container. Compute resources in the context of Kubernetes are defined as something that can be requested by, allocated to and consumed from a container. The resources are categorized as compressible (one that can be throttled such as CPU, network bandwidth) and incompressible (cannot be throttled, such as memory). It is important to make the distinction between compressible and incompressible resources, as if your containers consume too much compressible resources (such as CPU) they will be throttled, but if they consume too much incompressible resources (such as memory) they will be killed (as there is no other way to ask an application to release allocated memory).



Runtime resource demands

Based on the nature and the implementation details of your application, you have to specify the minimum amount of resources that are needed (called request) and the maximum amount it can grow up to (the limit). At high level, the concept of request/limit is similar to soft/hard limits. For example, in a similar manner we define heap size for a Java application using Xms and Xmx. So it makes sense in practice to define the request as a fraction of the limit of a container resource.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      limits:
        cpu: 300m
        memory: 200Mi
      requests:
        cpu: 200m
        memory: 100Mi
    terminationMessagePath: /dev/termination-log
```

Depending on whether you specify the request, the limit, or both, the platform offers different kind of Quality of Service (QoS).

- Best-Effort pod is the one that does not have requests and limits set for its containers. Such a pod is considered as lowest priority and will be killed first when the node the pod is placed on runs out of incompressible resource.
- Burstable pod is the one that has request and limit defined (or defaulted) but they are not equal (limit > request as expected). Such a pod has minimal resource guarantees, but also willing to consume more resources up to its limit when available. When the node is under incompressible resource pressure, these pods are likely to be killed if there are no Best-Effort pods remaining.
- Guaranteed pod is the one that has equal amount of request and limit resources. These are highest priority pods and guaranteed not to be killed before Best-Effort and Burstable pods.

So the resource characteristics you defined or omit for the containers have direct impact on its QoS and defines the relative importance of the pod in the time of

resource starvation. Define your pod resource requirements with its consequences in mind.

Project Resources

Kubernetes is a self-service platform that enables developers to run applications as they see suitable on the designated isolated environments. But working in a shared multi-tenanted platform requires also the presence of certain boundaries and control units to prevent few users consuming all the resources of the platform. One such a tool is ResourceQuota which provides constraints to limit the aggregated resource consumption in a namespace. With ResourceQuotas, the cluster administrators can limit the total sum of compute resources (cpu, memory) and storage. It can also limit the total number of objects (such as configmaps, secrets, pods, services, etc) created in a namespace. Another useful tool in this area is LimitRange which allows setting resource usage limits for each type of resource in a namespace. In addition to specifying the minimum and maximum allowed amounts for different resource types and the default values for these resources, it also allows you to control the ratio between the request and limit (overcommit levels).

Type	Resource	Min	Max	Default Limit	Default Request	Lim/Req Ratio
Container	cpu	0.5	2	500m	250m	4
Container	memory	250Mi	2Gi	500Mi	250Mi	4

LimitRanges are useful for controlling the container resource profiles so that there are no containers that require more resources than a cluster nodes can provide. It can also prevent from creating containers that consume large amount of resources and making the nodes not allocatable for other containers. Considering that the request (and not limit) is the main container characteristic the scheduler uses for placing, LimitRequestRatio allows you control how much difference there is between the request and limit of containers. Having a big combined gap between request and limit will increase the chances for overcommiting on the node and may degrade the application performance when many containers require at the same time more than requested amounts of resources.

Capacity Planning

Considering that containers may have different resource profiles on different environments, it becomes obvious that capacity planning for a multi-purpose environment is not straightforward. For example for best hardware utilization, on a non-production cluster you may have mainly BestEffort and Burstable containers. On such a dynamic environment there will be many containers that are starting up and shutting down at the same time and even if a container gets killed by the platform during resource starvation, it is not fatal. On the production cluster where we want things to be more stable and predictable, the containers may be mainly from Guranteed type and some Burstable. And if a container gets killed, that most likely will be a sign to increase the capacity of the cluster.

Here is an example table where we have few services with CPU and Memory demands.

Pod Name	CPU Request	CPU Limit	Memory Request	Memory Limit	Instances
Service A	500m	500m	500Mi	500Mi	4
Service B	250m	500m	250Mi	1000Mi	2
Service C	500m	1000m	1000Mi	2000Mi	2
Service D	500m	500m	500Mi	500Mi	1
Total	4000m	5500m	5000Mi	8500Mi	9

Of course in a real life scenario, the reason why you are using a platform such as Kubernetes will be because there are many more services, some of which will be about to retire, and some will be still at design and development phase. Even if it is a constantly moving target, based on a similar appraoche as above, we can calculate the total amount of resources needed for all the services per environment. Keep in mind that on the different environments there will also different number of container instances and you may even need to leave some room for autoscaling, build jobs, infrastructure containers, etc. Based on this information and the the infrastructure provider, then you can choose the most cost effective compute instances that provide

the required resources.

Discussion

Containers are not only good for process isolation and as a packaging format, but with identified resource profiles, they are also the building block for a successful capacity planning. Perform some tests early and discovery what are the resources needed for each container and use that as a base for future capacity planning and prediction.

More Information

Programming language benchmarks
Using ConfigMap
Best Practices for Configuration by Kubernetes Persistent Volumes by Kubernetes
Resource Quotas by Kubernetes
Applying Resource Quotas and Limits by Kubernetes
Resource Quality of Service in Kubernetes by Kubernetes
Setting Pod CPU and Memory Limits by Kubernetes

3. Dynamic Placement

Allows applications to be placed on the cluster in a predictable manner based on application demands, available resources and guiding policies.

Problem

A good sized Microservices based system will consist of tens or even hundreds of isolated processes. Containers and Pods do provide nice abstractions for packaging and scaling, but does not solve the problem of placing these processes on the suitable hosts. With a large and ever growing number of Microservices, assigning and placing them individually to hosts is not a manageable activity. Containers have dependencies among themseves, dependencies to hosts, resource demands, and all of that changes over time too. The resources available on a cluster also vary over time, through shrinking or extending the cluster, or by having it consumed by already placed containers. The way we place containers also impacts the availability, performance, and capacity of the distributed systems as well. All of that makes placing containers to hosts a moving target that has to be shot on the move.

Solution

In this chapter we will not go into the details of scheduler as this is an area of Kubernetes that is highly configurable and still evolving rapidly. However, we will cover what are the driving forces that affect the placement, why chose one or the other option, and the resulting consequences. Kubernetes scheduler is a very powerful and time saving tool. It plays a fundamental role in the Kubernetes platform as a whole, but similarly to other components (API Server, Kubelet) it can be run in isolation or not used at all. At a very high level, the main operation it performs is to retrieve each newly created pod definition from the API Server and assign it to a node. It finds a suitable host for every pod to be run(as long as there is such

a host) whether that is for the initial application placement, scaling up, or when moving an application from a unhealthy host ot a healthier one. And it does it by considering runtime dependencies, resource requirements, and guiding policies for HA by spreading pods horizontally, and/or also for performance and low latency interactions by co-locating pods nearby. But for the scheduler to do its job properly and allow declarative placement, it needs the following three parts to be in place.

Available Node Resources

To start with, the Kubernetes cluster needs to have nodes with enough resource capacity to run the the pods. Every node has a maximum resource capacity available for running pods and the scheduler ensures that the sum of the resources requested for a pod is less than the available allocatable node capacity. Considering a node dedicated only to Kubernetes, its capcity is calculated using the formula: [Allocatable] = [Node Capacity] - [Kube-Reserved] - [System-Reserved]. Also keep in mind that if there are containers running on the node that are not managed by Kubernetes, that will not be reflected in the node capacity calculations by Kubernetes. A workaround for this limitation is to run a placeholder pod which doesn't do anything, but only has resource limit corresponding to the not tracked containers' resource usage amount. Such a pod is created only to represent the resource consumption of the not tracked containers and helps the scheduler to build a better resource model of the node.

Container Resources Demands

The second piece is having containers with runtime dependencies and resource demands defined. We have covered that in more details in the Predictable Demands chapter. And it boils down to having containers that declare their resource profiles (with request and limit) and environment dependencies such as storage, ports, etc. Only then, pods will be assigned to hosts in a sensible way and they will be able to run without affecting each other during peak times.

Placement Policies

The last piece of the puzzle is having the right filtering or priority policies for your specific application needs. The scheduler has a default set of predicate and priority

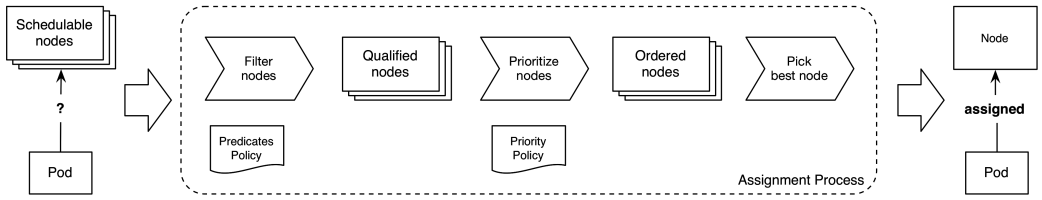
policies configured that is good enough for most use cases. That can be overridden during scheduler startup with a different set of policies such as the following one:

```
{
    "kind" : "Policy",
    "apiVersion" : "v1",
    "predicates" : [
        {"name" : "PodFitsPorts"},
        {"name" : "PodFitsResources"},
        {"name" : "NoDiskConflict"},
        {"name" : "NoVolumeZoneConflict"},
        {"name" : "MatchNodeSelector"},
        {"name" : "HostName"}
    ],
    "priorities" : [
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 1},
        {"name" : "EqualPriority", "weight" : 1}
    ]
}
```

Notice that in addition to configuring the policies of of the default scheduler, it is also possible to run multiple schedulers and allow pods tho specify which scheduler to place them. You can start another scheduler instance that is configured differently by giving it a unique name. Then when defining a pod add an annotation such as the following: *scheduler.alpha.kubernetes.io/name: my-custom-scheduler* and the pod will be picked up by the custom scheduler.

Scheduling Process

Good sized pods get assigned to nodes by guiding policies. For completeness, let's visualize at high level how these elements get together and the main steps a pod scheduling goes through:



A pod to node assigment process

As soon as a pod is created that is not assigned to a host yet(nodeName=""), it gets picked by the scheduler together with all the schedulable hosts (schedulable=true) and the set of filtering and priority policies. In the first stage, the scheduler applies the filtering policies and remove all nodes which do not qualify based on the pod criteria. In the second stage, the remaining hosts get ordered by weight. And in the last stage the pod gets a node assigned which is the main change that happens during scheduling.

Forcing Assignment

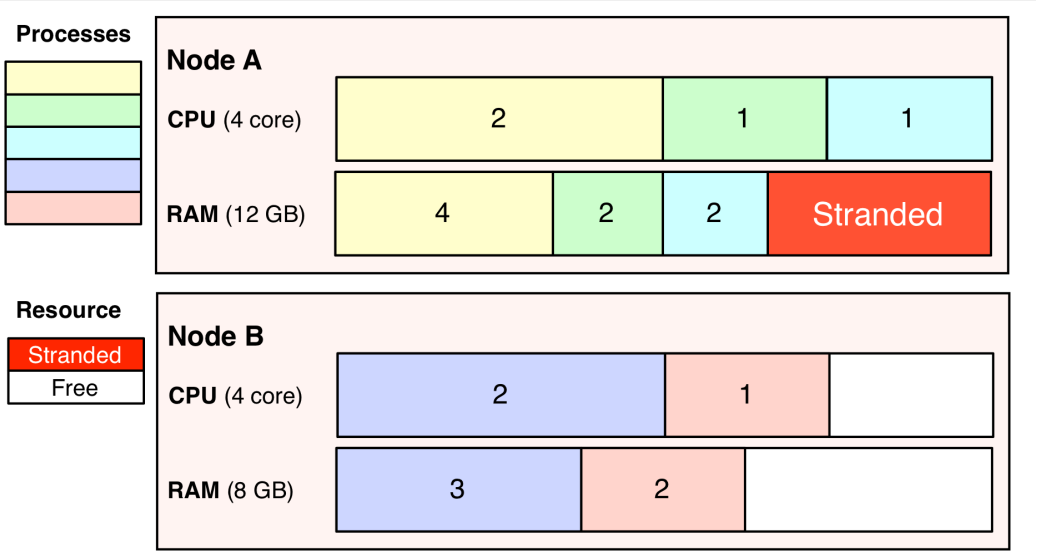
For most of the cases it is better to let the scheduler do the pod to host assignment and not micro manage the placement logic. But in some rare occasions, you may want to force the assignment of a pod to a specific host or to a group of hosts. This can be done using the nodeSelector. NodeSelector is part of PodSpec and specifies a map of key-value pairs that must be present on the node in order for the node to be eligible to run the pod. An example here would be if you want to force a pod to run on some specific host where you have SSD storage or GPU acceleration hardware. With the following pod definition that has nodeSelector matching disktype: ssd, only nodes that are labelled with disktype=ssd will be eligible to run the pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

In addition to adding custom labels to your nodes, you can also use some of the default labels that are present on every node. Every host will have a unique kubernetes.io/hostname label that can be used to place a pod on a host by its hostname. There are other default labels which indicate the OS, architecture, instance-type that can be useful for placement too.

Discussion

Placement is an area where you want to have as minimal interventions as possible. If you follow the guidelines from Predictable Demands chapter and declare all the resource needs of a container, the scheduller will do its job and place the container on the most suitable node possible.



Processes scheduled to hosts and stranded resources

You can influence the placement based on the application HA and performance needs, but try not to limit the scheduler to much and place yourself into a corner where no more Pods can be scheduled and there are to much stranded resources.

More Information

Assigning Pods to Nodes by Kubernetes
Running in Multiple Zones by Kubernetes
Node Placement and Scheduling Explained
Pod Disruption Budget by Kubernetes
Guaranteed Scheduling For Critical Add-On Pods by Kubernetes
The Kubernetes Scheduler by Kubernetes
Scheduler Algorithm by Kubernetes
Configuring Multiple Schedulers by Kubernetes
Node Allocatable Resources by Kubernetes
Everything You Ever Wanted to Know About Resource Scheduling, But Were Afraid to Ask by Tim Hockin

4. Declarative Deployment

The deployment abstraction encapsulates the upgrade and rollback process of a group of containers and makes executing it a repeatable and automatable activity.

Problem

We can provision isolated environments as namespaces in self service manner, and have the services placed on these environments with minimal human intervention through the scheduler. But with a growing number of microservices, updating and replacing them with newer versions constantly becomes a growing burden too. Luckily, Kubernetes has thought and automated this activity too. Using the concept of Deployment we can describe how our application should be updated, using different strategies, and tuning the different aspects of the update process. If you consider that you do multiple deployments for every Microservice instance per release cycle (for some this is few minutes, for some it is few months), this is a very effort-saving automation.

Solution

We have seen that to do its job effectively, the Scheduler requires suffitient resources on the host system, appropriate placement policies, and also containers with properly defined resource profiles. In a similar manner, for a Deployment to do its job properly, it expects the containers to be a good cloud native citizens. At the very core of a Deployment is the ability to start and stop containers in a controlled manner. For this to work as expected, the container itself has to listen and honour lifecycle events (such as SIGTERM) and also provide health endpoints indicating whether it has started successfully or not. If a container covers these two areas properly, then the platform can cleanly shut down running container instances and replace them by starting new healthy instances. Then all the remaining aspects of an update

process can be defined in a declarative way and executed as one atomic action with predefined steps and expected outcome. Let's see what considerations are there for a container update behavior.

Imperative kubectl rolling-update

Typically, containers are deployed through Pods which are created and managed by a ReplicationController or some other control structure. One way to update a group of containers without any downtime is through kubectl rolling-update command. A command such as the following:

```
$ kubectl rolling-update frontend --image=image:v2
```

performs changes to the running container instances in the following stages:

- Creates a new ReplicationController for the new containers instances.
- Decreases the replica count on the old ReplicationController.
- Increases the replica count on the new ReplicationController.
- Iterates over the above two steps until all container instances are replaced with the new version.
- Removes the old ReplicationController which should have replica count of zero at this stage.

As you can imagine there are many things that can go wrong during such an update process. For this reason, there are multiple parameters to control the various timings for the process, but more importantly, there is also the rollback option to use when things go wrong and you want to recover the old state of the containers.

Even though kubectl rolling-update automates many of the tedios tasks for container update and reduces the number of commands to issue to one, it is not the preffered approach for repeatable deployments. Here are few reasons why:

- Imperative nature of the update process triggered and managed by kubectl. The whole orchestration logic for replacing the containers and the ReplicationControllers is implemented in and performed by kubectl which interacts with the API Server behind the scene while update process happens.

- You may need more than one command to get the system into the desired state. These commands will need to be automated and repeatable on different environments.
- Somebody else may override your changes with time.
- The update process has to be documented and kept up to date while the service evolves.
- The only way to find out what was deployed is by checking the state of the system. Sometimes the state of the current system might not be desired state, in which case it has to be correlated with the deployment documentation.

Next, let's see how to address these concerns by making the update process declarative.

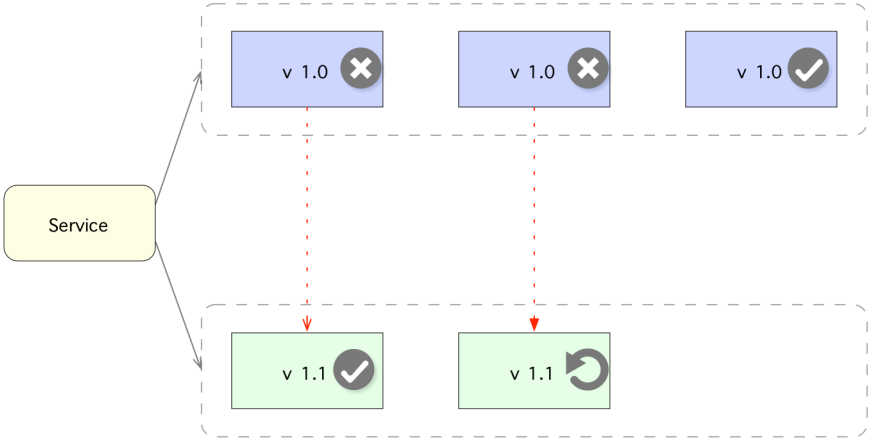
Rolling Deployment

The declarative way of updating applications in Kubernetes is through the concept of Deployment. So rather than creating a ReplicationController to run Pods and then issues kubectl rolling-update commands for updating, you can use the Deployment abstraction and define both of these aspects of your application. Behind the scenes the Deployment creates ReplicaSet which is the next-generation ReplicationController that supports set-based label selector. In addition, the Deployment abstraction also allows shaping the update process behaviour with strategies such as RollingUpdate (default) and Recreate.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

RollingUpdate strategy behavior is similar to kubectl rolling-update behaviour which ensures that there is no downtime during the update process. Behind the scene, the Deployment implementation performs similar moves by creating new ReplicaSet and replacing old container instances with new ones. One enhancement here is that with Deployment it is possible to control the rate of new container rollout. The kubectl rolling-update command would replace a single container at time, whereas Deployment object allows you to control that range through maxSurge maxUnavailable fields.



Rolling deployment

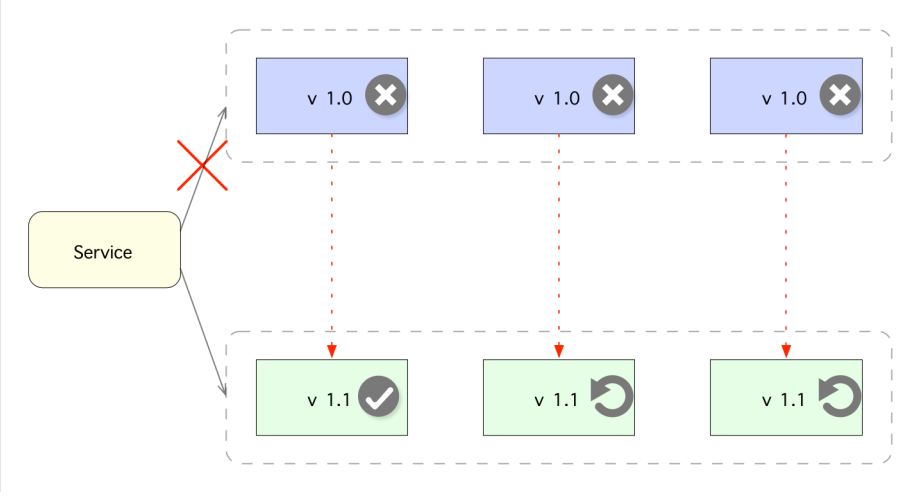
In addition to addressing the previously mentioned drawbacks of the imperative way of deploying services, the Deployments bring the following benefits:

- The Deployment is a server side object and the whole update process is performed on the server side.

- The declarative nature of Deployment makes you envisage how the deployed state should look like rather than the steps necessary to get there.
- The whole deployment process is an executable document, tried and tested on multiple environment, and maintained.
- It is also fully recorded, versioned with options to pause, continue and rollback to previous versions.

Fixed Deployment

RollingUpdate strategy is good for ensuring zero downtime during the update process. But the side effect of this strategy is that during the update process there will be two different versions of the container running at the same time. That may cause issues for the service consumers especially when the update process has introduced backword uncompatible changes in the service APIs. For this kind of scenarios there is the Recreate strategy.



Fixed deployment

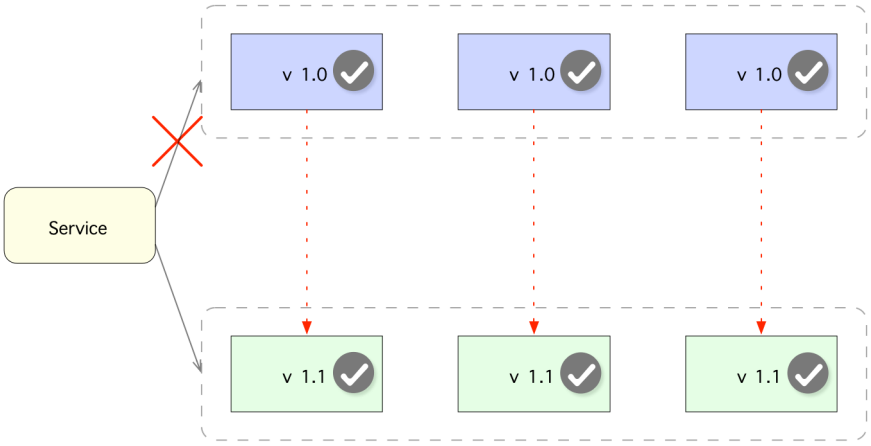
Recreate strategy basically has the effect of setting maxUnavailable=0, which means it first kills all containers from the current version, and then starts new containers. The result of this sequence of actions is that there will be some down time while all containers are stopped and there no new containers ready to handle coming requests. But on the positive side, it means there won't be two versions of the containers

running at the same time simplifying the life of service consumers to handle only one version at a time.

Blue-green Release

The Deployment is a fundamental concept that let's you define how immutable containers are transitioned from one version to another. We can use the Deployment primitive as a building block together with other Kubernetes primitives to implement more advanced release strategies such as Blue-green deployment and Canary Release. The Blue-green deployment is a release strategy used for deploying software in a production environment by minimizing downtime and reducing risk. A Blue-green deployment works by creating a new ReplicaSet with the new version of the containers (let's call it green) which are not serving any requests yet. At this stage the old container replicas (called blue) are still running and serving live requests.

Once we are confident that the new version of the containers are healthy and ready to handle live requests, we switch the traffic from old container replicas to the new cotnainer replicas. This activity in Kubernetes can be done by updating the service selector to match the new containers (tagged as green). Once all the trafic has been handled by the new green containers, the blue container can be deleted and resources utilized by future Blue-green deployment.

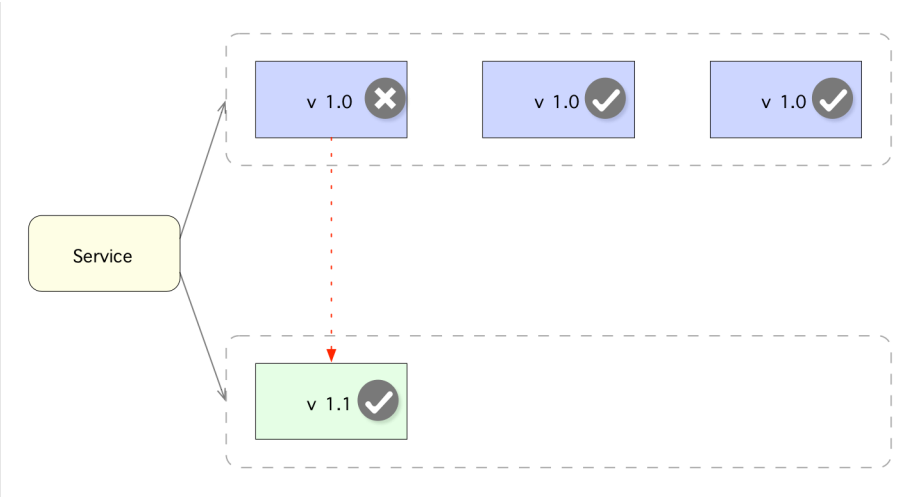


Blue-Green release

A benefit of Blue-green approach is that thereâ€™s only one version of the application serving requests, which reduces the complexity of handling multiple concurrent versions by the consumers. The down side is that it requires twice of the capacity of normal capacity while both blue andd green containers are up and running.

Canary Release

Canary Release is a way for softly deploying a new version of an application into production enviornment by replacing only small subset of old instances with new ones. This technique reduces the risk of introducing a new version into production by letting only a part of the consumers reach the updated version. When we are happy with the new version of our service and how it performed with the small sample of consumers, we replace all the old instances with the new version.

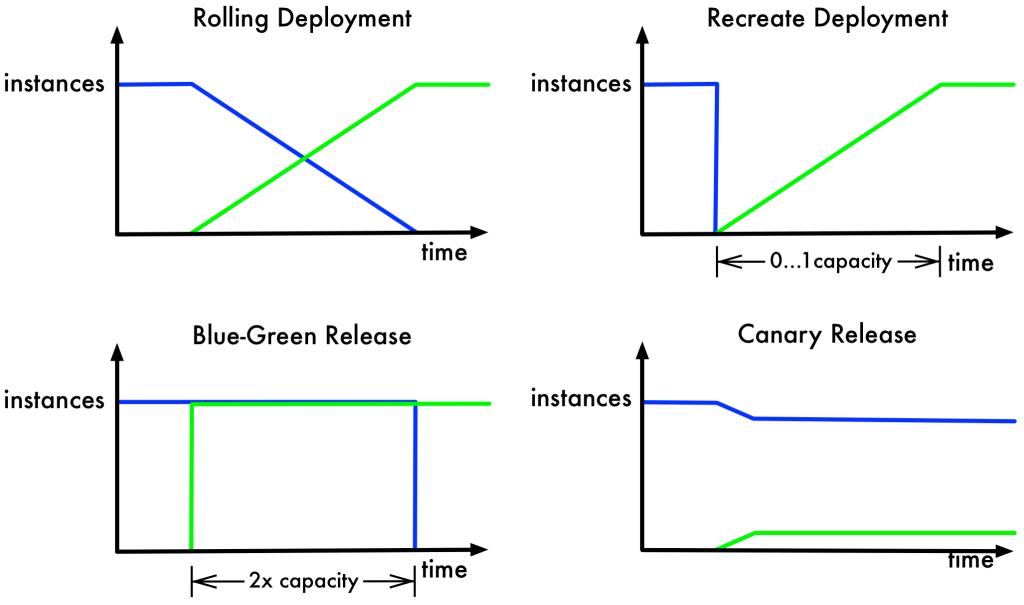


Canary release

In Kubernetes this technique can be implemented by creating a new ReplicaSet for the new container version, (preferably using a Deployment) with a small replica count that can be used as the Canary instance. At this stage, the service should direct some of the consumers to the updated pod instances. Once we are confident that everything with new ReplicaSet works as expected, we scale new ReplicaSet up and the old ReplicaSet to zero. In a way we are performing a controlled and user tested incremental rollout.

Discussion

The Deployment primitive is an example where the tedious process of manually updating applications has been turned into a simple declarative action that can be automated. The OOTB deplyoment strategies (rolling and recreate) control how old container instanes are replaced by new ones, and the release strategies (blue-green and canary) control how the new version becomes available to service consumers. The release strategies are not fully automated and require human intervention, but that may change in near future.



Deployment and release strategies

Every software is different, and deploying complex systems usually requires additional steps and checks. As of this writing, there is a proposal for Kubernetes to allow hooks in the deployment process. Pre and Post hooks would allow the execution of custom commands before and after a deployment strategy is executed. Such commands could perform additional actions while the deployment is in progress and would additionally be able to Abort, Retry, Continue a deployment. A good step that will give birth to new automated deployment and release strategies.

More Information

Rolling Update Replication Controller by Kubernetes
Deployments by Kubernetes
Deploying Applications by Kubernetes
Blue-Green Deployment by Martin Fowler
Canary Release by Martin Fowler
DevOps with OpenShift by S. Picozzi, M. Hepburn, N. Oâ€™Connor

5. Observable Interior

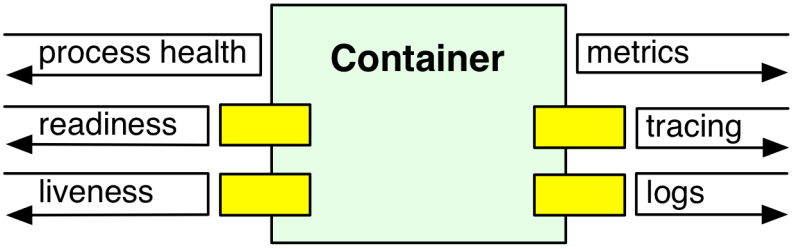
In order to be fully automatable, cloud native applications must be highly observable by providing some means to the managing platform to read and interpreset the application health and if necessary take mitigative or corrective actions.

Problem

Kubernetes will regularly check the container process status and restart it if issues are detected. But from practice we know that it is not enough to judge about the health of an application based on process status. There are many cases where an application is hung but its process is still up and running. For example a Java application may throw an OutOfMemoryError and still have the JVM process up. Or an application may freeze because it run into an infinite loop, deadlock or some kind of thrashing (cache, heap, process). To detect these kinds of situations, Kubernetes needs a reliable way to check the health of the applications. That is not to understand how an application works internally, but a check that communicates whether the application is functioning properly and capable to serve consumers.

Solution

The software industry has accepted the fact that it is not possible to write bug-free code. And the chances for failure increases even more when working with distributed applications. As a result the the focus for dealing with failures has shifted from avoiding them, to detecting failures and recovering.



Container observability APIs

Detecting failure is not a simple task that can be performed uniformly for all applications as all applications have different definition of a failure. Also there are different type of failures that require different corrective actions. Transient failures may self-recover given enough time, and some other failures may need a restart of the application. Let's see the different checks Kubernetes uses to detect and correct failures.

Process Health Checks

Process health check is the simplest health check that the Kubelet performs constantly on the container processes. And if the container processes is not running, it will be restarted. So even without any other health checks, the application becomes slightly more robust though this generic checks. If your application is capable of detecting any kind failures and shutting itself down, the process health check is all you need. But for the most cases that is not enough and other types of health checks are also necessary.

Liveness Probes

If your application runs into some kind of DeadLock, it will still be considered healthy from process health check point of view. To detect this kind of issues and any other types of failures according to your application business logic, Kuberntes has the concenpt of Liveness probe. This is a regular check performed by the Kubelet agent that asks your container to confirm it is still healthy. It is important to have the healthcheck performed from the outside rather than the application itself as some

failures may prevent the application watchdog to report its own failure. In terms of corrective action, this health check is similar to process health check as if a failure is detected, the container is restarted. But it offers more flexibility in terms of what methods to use for checking the application health:

- HTTP probe performs an HTTP GET request to the container IP address and expects a successful HTTP response code (2xx or 3xx).
- A TCP Socket probe expects a successful TCP connection.
- An Exec probe executes an arbitrary command in the container namespace and expects successful exit code (0).

An example HTTP based Liveness probe is shown below:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-liveness-check
spec:
  containers:
  - name: nginx
    image: nginx
    livenessProbe:
      httpGet:
        path: /_status/healthz
        port: 80
      initialDelaySeconds: 30
      timeoutSeconds: 1
```

Depending on the nature of your application, you can chose the method that is most suitable for you. And it is up to your implementation to decide when your application is considered healthy or not. But keep in mind that the result of not passing a health check will be restating of your container. If your container is failing for some reason that restarting will not help, then there is not much benefit in having a failing liveness check as your container will be restated without recovering.

Readiness Probes

Liveness checks are useful for keeping applications healthy by killing unhealthy containers and replacing them with new ones. But sometimes a container may not be healthy and restarting it may not help either. The most common example is when a container still starting up and not ready to handle any requests yet. Or may be a container is overloaded and its latency is increasing, and you want to it to shield itself from additional load for a while. For this kind of scenarios, Kubernetes has Readiness probes. The methods for performing readiness check are the same as Liveness checks (HTTP, TCP, Exec), but the corrective action is different. Rather than restarting the container, a failed Readiness probe causes the container to be removed from the service endpoint and not recieve any new traffic. Readiness probes give some breathing time to a starting container to warm up and get ready before being hit with requests from the service. It is also useful for schielding the service from traffic at later stages as readiness probes are performed regularly similarly to liveness check.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
spec:
  containers:
  - name: nginx
    image: nginx
    readinessProbe:
      exec:
        command:
        - ls
        - /var/ready
```

Again, it is up to your implementation of the health check to decide when your application is ready to do its job and when it should be left alone. While process health check and liveness check are intended to actively recover from the failure by restarting the container, the readiness check buys time for your application and expect it to recover by itself. Keep in mind that Kubernetes will try to prevent

your container from receiving new requests when it is shutting down for example regardless if the readiness check still passes or not after receiving a SIGTERM signal. In many cases you will have Liveness and Readiness probes performing the same check. But the presence of readiness probe will give some time your container to start up and only passing the readiness probe will consider a deployment successful and trigger the killing of other containers as part of a rolling update for example. The Liveness and the Readiness probes are fundamental building blocks in the automation of cloud native applications. These probes exists in other platforms such as Docker, Apache Mesos and implementation are provided in my application frameworks (such as Spring actuartor, WildFly Swarm healthck, MicroProfile spec for Java).

Discussion

Health checks play a fundamental role in the automation of various application related activities such as deployment and self-healing. But there are also other means through which your application can provide more visibility about its health. The very obvious and old method for this purpose is through logging. It is a good practice for containers to log any significant events to system out and system error and have these logs collected to a central location for further analysis. Logs are not typically used for taking automated actions, but rather to raise alerts and further investigations. A more useful aspect of logs is the post mortem analysis of failures and detecting unnoticeable errors.

Apart from logging to standard streams, it is also a good practice to log the reason for exiting a container to `/dev/termination-log`. This location is the place where the container can do its last will before being permanently vanished.

More Information

Configuring Liveness and Readiness Probes by Kubernetes
Working with Containers in Production by Kubernetes
Graceful shutdown with Node.js and Kubernetes
Readiness Checks by Marathon
Health Checks and Task Termination by Marathon

6. Life Cycle Conformance

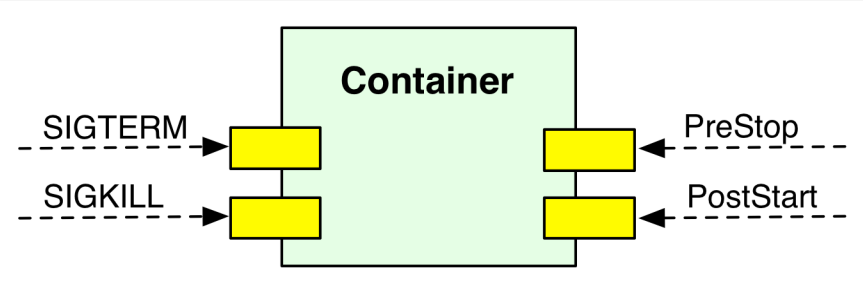
Containerized applications managed by cloud native platforms have no control over their life cycle and in order to be good cloud native citizens they have to listen to the events emitted by the managing platform and conform their life cycles accordingly.

Problem

In the Observability chapter we explained why containers have to provide APIs for the different health checks. Health check APIs are read only endpoints that the platform is probing constantly to get some application insight. It is a mechanism for the platform to extract information from the application. In addition to monitoring the state of a container, the platform sometimes may issue commands and expect the application to react. Driven by policies and external factors, a cloud native platform may decide to start or stop the applications it is managing at any moment. It is up the containerized application to decide which events are important to react and how to react. But in an effect, this is an API that the platform is using to communicate and send commands to the application. And the applications are free to implement these APIs if they want to provide a greater consumer service experience or ignore.

Solution

We saw tha only checking the process status is not good enough indication for the health of an application. That is why there are different APIs for checking the health of a container. Similarly, using only the process model to run and kill a process is not good enough. Real world applications require more fine grained interactions and life cycle management capabilities for a greater consumer experience. Some applications need a kick in the butt to warm up, and some applications need a gentle and clean shut down procedure. For this and other use cases there are events that are emitted by the platform which the container can listen to and react if desired.



Life Cycle Conformance

An application or a microservice corresponds to the deployment and management unit of pod. As we already know, a pod is composed of one or more containers. At pod level, there are other constructs such as init-container and defer-containers (which is still at proposal and development stage as of this writing) that can help manage the pod life cycle. The events and hooks we describe at this chapter are all applied at individual container level rather than pod.

SIGTERM Signal

Whenever a container has to shut down, whether that is because the Pod it is belonging to is shutting down, or simply a failed livenes probe causes the container to be restarted, the container will receive a SIGTERM signal. SIGTERM is a gentle poke for the container to shut down cleanly before a more abrupt SIGKILL signal is sent. If the application in the container has been started as part of the container main process and not as a child process, the application will receive the signal too (which is the expected behaviour). Once a SIGTERM signal is received the application should as quickly as possibly shut down. For some applications this might be a quick termination, and some other applications may have to complete their in flight requests, release open connections, cleanup temp files, which can take slightly longer time. In all cases, reacting to SIGTERM is the right moment to shut down a container in a clean way.

SIGKILL Signal

If an container process has not shutdown after SIGTERM signal, it is shut down forcefully by the following SIGKILL signal. SIGKILL is not issued immediately, but

a grace period of 30 seconds by default is waited after SIGTERM. This grace period can be defined per container basis (using terminationGracePeriodSeconds field), but cannot be guaranteed as it can be overriden while issusing commands to Kubernetes. The preffered approach here is to design and implement containerized applications to be ephemeral with quick startup and shut down process.

Post-start Hook

Using only process signals for providing rich application life cycle management experience is somewhat limited. That is why there are additional life cycle hooks such as postStart and preStop provided by Kubernetes that you can plug your application into. An extract of pod manifest containing a postStart hook looks like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
  containers:
  - image: some-image
    name: main
    lifecycle:
      postStart:
        exec:
          command:
          - /bin/postStart.sh
```

PostStart hook is executed after a container is created, asynchronously with the main container process. Even if many of the application initialization and warm up logic can be implemented as part of the container startup steps, PostStart still covers some use cases. PostStart is a blocking call and the container status remains Waiting until the postStart handler completes which in turn keeps pod status in Pending state. This nature of PostStart can be used to delay the startup state of container while giving time to the main container process to initialize. Another use of PostStart is to prevent a container to start when certain preconditions are not met. For example a failure in

running PostStart hook or non-zero exit code will cause the main container process to be killed.

PostStart and PreStop hooks invokation mechanisms are similar to healthcheck probes and support two handler types: - Exec - which runs a command inside the container; - HTTP - which executes and HTTP request against the container;

You have to be very careful what critical logic you execute in PostStart hook as there are not many guarantees for its execution. Since the hook is running in parallel with the container process, it is possible that the hook is executed before the container has started. In addition the hook is intended to have "at least once" semantics, so the implementation has to take care of duplicate executions. Another aspect to keep in mind is that the platform is not performing any retry attempts on failed HTTP requests that didn't reach the handler.

Pre-stop Hook

PreStop hook is a blocking call, send to a container before it is terminated. It has the same semantics as SIGTERM signal and should be used to initiate a graceful shutdown of the container when catching SIGTERM is not possible.

```
apiVersion: v1
kind: Pod
metadata:
  name: pre-stop-hook
spec:
  containers:
  - image: some-image
    name: main
    lifecycle:
      preStop:
        httpGet:
          port: 8080
          path: shutdown
```

PreStop hook must complete before the call to delete the container is sent to Docker doemon which triggers SIGTERM notification. Even though PreStop is a blocking

call, holding on it or returning a non-succesful result will not prevent the container from being deleted and the process killed. PreStop is only a convinient alternative for SIGTERM signal for graceful application shutdown and nothing more. It also offers the same handler types and guarantees as PostStart hook we covered previously.

Discussion

One of the main benefits the cloud native platforms provide is the ability to run and scale applications reliably and predictable on top of potentially unreliable cloud infrastructure. To achieve that, these platforms offer a set of contracts and impose a set of constraints on the applications to conform with. It is in the interest of the application to confrom to these contracts (lifecycle events are a good example) in order to benefit from all of the capabilities offered by the cloud native platform. Conforming to these events will ensure that your application have ability to gracefully start up and shut down with minimal impact on the consuming services. In the future there might be even more events giving hints to the application when it is about to be scaled up, or asked to release resources to prevent being shut down etc. It is important to get into the mindset where the application lifecycle is not any longer in the control of Ops team, but fully automated by the platform.

More Information

Container Lifecycle Hooks by Kubernetes
Attaching Handlers to Container Lifecycle Events
Graceful shutdown of pods with Kubernetes

II Behavioral Patterns

The patterns under this category are focused more around common communication mechanisms between containers and also the managing platform itself.

7. Batch Job

The Job primitive allows performing a unit of work in a reliable way until completion.

Problem

The main primitive in Kubernetes for managing and running containers is the pod. There are different ways for creating pods with varying characteristics:

- Bare pod: it is possible to create a pod manually to run containers. But when the node such as pod is running on fails, the pod will not be restarted.
- ReplicationController/ReplicaSet: these controllers are used for creating multiple pods that are expected to run continiusly and not terminate (for example to run an httpd container).
- DaemonSet: a controller for running a single pod on every node.

A common theme among these pods is the fact that they represent long-running processes that are not meant to stop after some time. However in some cases there is a need to perform a predefined finite unit of work in reliable way and then shut down the application. This is what Kubernetes Jobs are primarily useful for.

Solution

A Kubernetes Job is similar to a ReplicationController as it creates one or more pods and ensures they run sucessfully. But the diffeerence is that, once a specific number of pods terminate succesfully, the Job is considered completed and pods are not scheduled or restarted any further. A Job definition looks like the following:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure
```

One important differences in Job and ReplicationController definition is the restartPolicy. The default restartPolicy for a ReplicationController is Always which makes sense for long-running processes that must be kept running. The value Always is not allowed for a Job and the only possible options are either OnFailure or Never.

One may ask why bother creating a Job to run a container until completion? Seems like an overkill, isn't it? There are a number of reliability and scalability benefits of using Jobs that might be useful:

- A container in a pod may fail for all kind of reasons and terminate with a non-zero exit code. If that happens and restartPolicy = "OnFailure", the container will be re-run.
- A pod may fail for a number of reasons too. For example it might be kicked out of the node or a container in the pod may fail and has restartPolicy = "Never". If that happens, the Job controller starts a new pod.
- If the node fails, the scheduler will place the pod on a new healthy node and re-run.
- A job will keep creating new pods forever if pods are failing repeatedly. It that is not the desired behaviour, a timeout can be set using the activeDeadlineSeconds field.

- A Job is not an ephemeral in-memory task, but a persisted one that survives cluster restarts.
- When a Job is completed, it is not deleted but kept for tracking purposes. The pods that are created as part of the Job are also not deleted but available for examination too.
- A Job may need to be performed multiple times. Using spec.completions field it is possible to specify how many times a pod should complete successfully before the Job itself is done.
- When a Job has to be completed multiple times (set through spec.completions), it can also be scaled and executed by starting multiple pods at the same time. That can be done in a declarative fashion by specifying spec.Parallelism field, or the imperative way using kubectl: kubectl scale –replicas=$N jobs/myjob

Job Types

From parallelism point of view, there are three types of jobs: - Non-parallel Jobs: this is the case when you leave both .spec.completions and .spec.parallelism unset and defaulted to 1. Such a Job is considered completed as soon as the pod terminates successfully. - Parallel Jobs with a fixed completion count: this is the case when .spec.completions is set to a number greater than 1. Optionally, you can set .spec.parallelism, or leave it unset to the default value of 1. Such a Job is considered completed when when there are .spec.completions. number of pods completed successfully. - Parallel Jobs with a work queue: this is the case when you leave .spec.completions unset (default to .spec.Parallelisms), and set .spec.parallelism to a an integer greater than 1. The way such a Job is considered completed is not very elegant though. A work queue job is considered completed when at least one pod has terminated successfully and all other pods have terminated too.

Usage Patterns

As you can see, the Job abstraction is a very powerful primitive that can turn work items into reliable parallel execution of pods. But it doesn't dictate how you should map individually processable work items into jobs and pods. That is something you have to come up considering the pros and cons of each option:

- One Job per work item: this options has some overhead to creates objects, and also for the system to manager large number or Jobs that consumer resources. This option is good when each work item is a complex task that has to be recorded, tracked, scaled, etc.
- One Job for all work items: this option is good for large number of work items as it has smaller resource demand on the system. It allows also scaling the Job easily through kubectl scale command.
- One Pod per work item: this option allows creating single purposed pods that are only concerned with processing a single work item to completion.
- One Pod for multiple work items: this option scales better as the same pod is reused for processing multiple work items.

More Information

Run to Completion Finite Workloads by Kubernetes
Parallel Processing using Expansions by Kubernetes
Coarse Parallel Processing Using a Work Queue by Kubernetes
Fine Parallel Processing Using a Work Queue by Kubernetes

8. Scheduled Job

A scheduled Job adds a time dimention to the Job abstraction allowing the execution of a unit of work to be triggered by a temporal event.

Problem

In the world of distributed systems and microservices, there is a clear tendency towards real time and event driven application interactions, using HTTP or light weight messaging. But job scheduling has a long history in software and regardless of the latest trends in IT, it is still as relevant as always has been. Scheduled Jobs are more commonly used for automating system maintenance or administration tasks. However, they are also relevant to application development where certain tasks need to run periodically. Typical examples here are data integration through file transfer, sending newsletter emails, backing up database, cleaning up and archiving old files, etc.

The traditional way of handling scheduled jobs for system maintenance purposes has been to use CRON. However, CRON jobs running on a single server are difficult to maintain and represent a single point of failure. For application based job scheduling needs, developers tend to implement solutions that are responsible to handle both the scheduling aspect and the job that has to be performed. In a similar fashion, the main difficulty with this approach is about making the scheduler scalable and highly available. Since the Job scheduler is part of the application, in order to make it highly available requires making the application itself highly available. That involves running multiple instances of the application, but at the same time ensuring that only a single instance is active and schedules jobs. Overall, a simple job such as one that copies files ends up requireing multiple instances and highly available storage mechnism. Kubernetes CronJob implementation solves all that by allowing scheduling Jobs using the well known CRON format.

Solution

In the previous chapter we saw what are the use cases for Jobs and what are capabilities provided by Kubernetes Jobs. All that applies for this chapter as well since Kubernetes CronJob primitive builds on top of Jobs.

A CronJob instance is similar to one line of crontap (cron table) and manages the temporal aspects of a Job. It allows the execution of a Job once in a future point of time or periodically at a specified points in time.

```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Apart from the Job spec, a CronJob has additional fields to define its temporal aspects:

- .spec.schedule takes a Cron format string as schedule time of its jobs to be created, e.g. 0 * * * *
- .spec.startingDeadlineSeconds stands for the deadline (in seconds) for starting the job if it misses its scheduled time. In some use cases there is no point in triggering a Job after certain time has passed the scheduled time.

- .spec.concurrencyPolicy specifies how to manage concurrent executions of jobs created by the same CronJob. The default behavaiour (Allow) will create new Job instances even if the previous Jobs have not completed yet. If that is not the desired behaviour, it is possible to skip the next run if the current one has not completed yet (Forbid) or it is also possible to cancel the currently running Job and start a new one (Replace).
- .spec.suspend field allows suspending all subsequent executions without affecting already started executions.
- .spec.successfulJobsHistoryLimit and .spec.failedJobsHistoryLimit fields specify how many completed and failed jobs should be kept for auditing purpose.

As you can see a CronJob is a pretty simple primitive that mainly brings CRON like behaviour. But when that is combined with the already existing primitives such as Job, Pod, Container and other Kubernetes features such as dynamic placement, health checks, etc it end up as a very powerful Job scheduling system. As a consequnece, developers can now focus on the problem domain and implement a containerized application that is only focused on the work item. The scheduling will be performed outside of the application, as part of the platform with all its added benefits such as high availability and resilience. Ofcourse, as with the Job implementation in general, when implementing CronJob, your application have to consider all corner cases of duplicate runs, no runs, parallel runs, cancelation, etc.

More Information

Cron Jobs by Kubernetes
Cron by Wikipedia

9. Daemon Service

Allows running infrastructure focused Pods on specific nodes, before application Pods are placed.

Problem

The concept of daemon in software systems exists at many levels. At operating system level, a daemon is a long-running computer program that runs as a background process. In Unix, the names of daemons end in "d" such as httpd, named, sshd. In other operating systems, alternative terms such as services, started tasks and ghost jobs are used. Regardless how they are called, the common characteristics among these programs is that they run as processes and usually do not interact with the monitor, keyboard, mouse and they are launched at system boot time. A similar concept exists at application level too. For example in the JVM there are daemon threads that run in the background and provides supporting services to the user threads. These daemon threads as are low priority, run in the background without a say in the life of the application, and performs tasks such as gc, finalizer etc. In a similar way, there is also the concept of DaemonSet at Kubernetes level. Considering that Kubernetes is a distributed system spread accross multiple nodes and with the primary goal of managing processes, a DaemonSet is represented by processes that run on these nodes and typically provide some background services for the rest of the cluster.

Solution

ReplicationController and ReplicaSet are control structures responsible for making sure that a specific number of pods are running. A ReplicationController, constantly monitors the list of running pods and makes sure the actual number of pods always matches the desired number. In that regards, DaemonSet is a similar construct which

is responsible for ensuring that a certain pods are always running. The difference is that the first two are tasked to run a specific number of pods determined by the expected load, irrespective of the node count - the typical application pod behaviour. On the other hand, a DaemonSet is not driven by consumer load in deciding how many pod instances to run and where to run. Its main tasks is to keep running a single pod on every node or a specific nodes. Let's see an example of such a DaemonSet definition next:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
 name: ssdchecker
spec:
 selector:
   matchExpressions:
     - key: checker
       operator: In
       values:
         - ssd
 template:
   metadata:
     labels:
       checker: ssd
   spec:
     nodeSelector:
       disk: ssd
     containers:
     - name: ssdchecker
       image: ssdchecker
```

Given this behaviour, the primary candidates for DaemonSet are usually infrastructure related process that perform cluster wide operations such as log collector, metric exporters, and even kube-proxy.

There are a number of differences how DaemonSet and ReplicationController or ReplicaSet are managed, but the main ones are as following:

- By default a DaemonSet will place one pod instance to every container, That can be controlled and limited to a subset of nodes using the nodeSelector field.
- A pod created by DaemonSet has already nodeName specified. As a resul the DaemonSet desn't require the existance of the Kubernetes scheduler to run containers. That also allows using DaemonSet for running and managing the Kubernets elements themselfvee.
- Pods created by DaemonSet can run before the scheduler has staterd, which allows them to run before any other services on a node.
- Since the scheduler is not used, the unschedulable field of a node is not respected by the DaemonSet controller.

Communicating with Daemon Pods

Given this characteristics of DaemonSets, there are also few specifics around communicating with Daemon Pods. Few possible ways for communicating with DaemonSets are:

- Push mechanism: the application in the DaemonSets pod pushes data to a well know location. No consumer reaches the DaemonSets pods.
- NodeIP and fixed port: pods in the DaemonSet can use a hostPort and become reachable via the node IPs. Consumers can reach every node by IP and the fixed port number.
- DNS: a headless service with the same pod selector as DaemonSet can be used to retrieve multiple A records from DNS containing all pod IPs and Ports.
- Service: create a service with the same pod selector as DaemonSet, and use the service to reach a daemon on a random node.

Even if the primary users of DaemonSets are likely to be cluster administrators, it is an interesting pattern for developers to be aware of.

More Information

Daemon Sets by Kubernetes
Performing a Rolling Update on a DaemonSet by Kubernetes
Daemon Sets and Jobs by Giant Swarm

10. Singleton Service

The implementations of Singleton Service pattern ensures that only one instance of a service is active at a time.

Problem

One of the main capabilities of Kubernetes is the ability to easily and transparently scale services. Pods can scale imperatively with a single command such as kubectl scale, or declaratively in a controller defition such as ReplicationController or ReplicaSet, or even dynamically scaled based on the application laod using HPA.

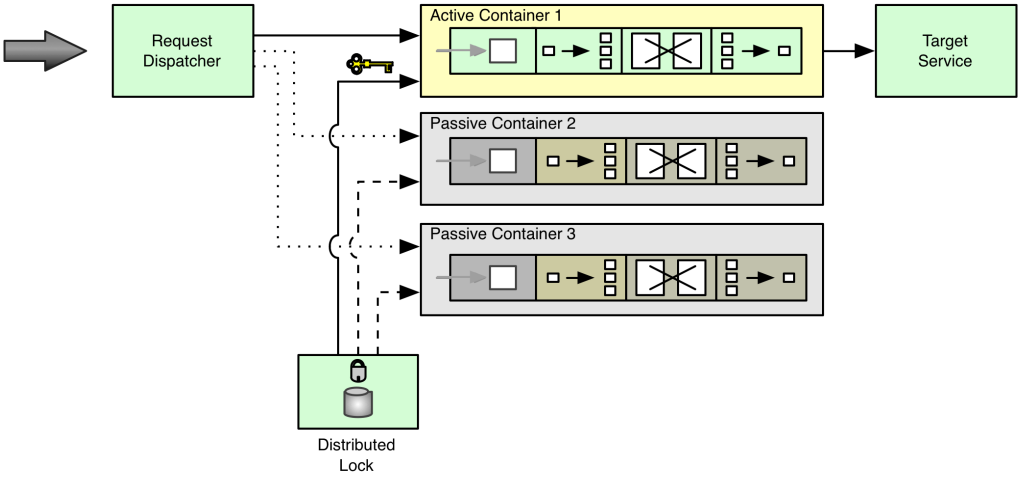
By running multiple instances of the same service, the system increases throughput and availability. The availability of the system increases as well because if an instance of a service becomes unavailable, the request dispatcher forwards future requests to healthy instances. But there are certain use cases where only one instance of a service is allowed to run at a time. For example, if we have a periodically executed job and run multiple instances of the same job, every instance would trigger at the scheduled intervals rather than having only one trigger fired as expected. Another example would be if the service performs polling on certain resources (a file system or database) and you want to ensure that only a single instance and maybe even a single thread performs the polling and processing. In all these and similar situations, you need some kind of control over how many instances (usually it is only one) of a service are active at a time, regardless of how many instances have been started.

Solution

Running multiple replicas of the same Pod creates an active/active topology where all instances of a service are active. What we need is an active/passive (or master/slave) topology where only one instance is active and all the other instances are passive. Fundamentally, this can be achieved at two possible levels: out-of-application locking and in-application locking.

Out-of-application Locking

As the name suggests, this mechanism relies on a managing process to ensure that only a single instance of the application is running. The application implementation itself is not aware that it is intended and will be run as a single instance. From this perspective it is similar to having an object in the Java world that is created as a singleton from the managing runtime such as Spring Framework.



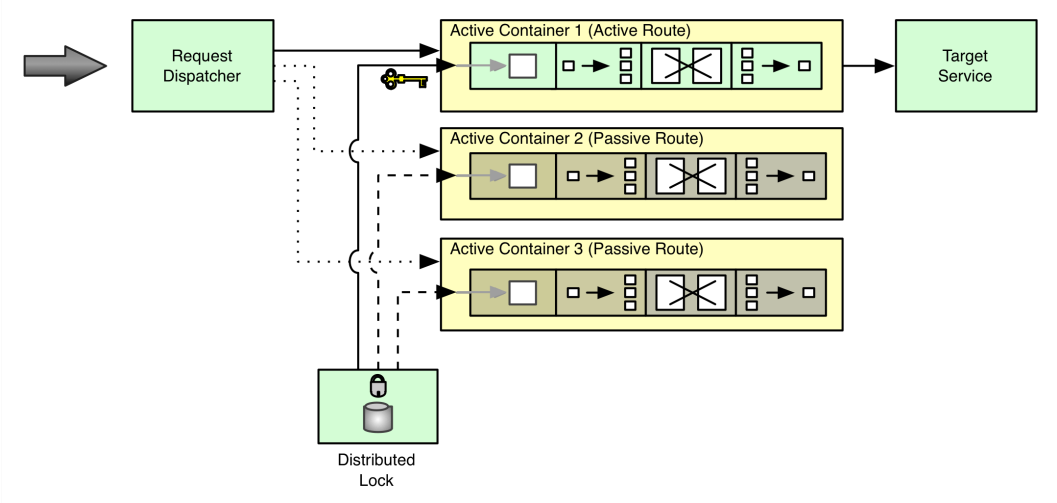
Out-of-application Locking

The way to achieve this in Kubernetes is simply to start a pod with one replica backed by a ReplicationController. Even if this topology is not exactly active/passive (there is no passive instance), it has the same effect, as Kubernetes ensures that there is only a single instance of the pod running at a time. In addition, the single pod instance is HA thanks to the ReplicationController and the health checks. The main thing to keep an eye on with this approach is the replica count which should not be increased accidentally.

In-application Locking

In a distributed system we can have control over the service instances through a distributed lock. Whenever a service instance is started, it will try to acquire the

lock, and if it succeeds, then the service will become active. Any subsequent service that fails to acquire the lock will wait and continuously try to get it in case the current active service releases this lock. This mechanism is widely used by many existing frameworks to achieve high availability and resilience. For example, Apache ActiveMQ can run in a master/slave topology where the data source is the distributed lock. The first ActiveMQ instance that starts up acquires the lock and becomes the master, and other instances become slaves and wait for the lock to be released.



In-application Locking

From a Java implementation perspective, this would be similar to the original Singleton Design Pattern from GoF where a class has a private constructor and a public static field that exposes the only instance of the class. So the class is written in a way that does not allow creation of multiple instances in a JVM. In the distributed systems world, this would mean, the service itself has to be written in a way that does not allow any more than one active instance at a time, regardless of the number of service instances that are started. To achieve this, first we need a central system such as Apache ZooKeeper or Etcd that provide the distributed lock functionality.

The typical implementation with ZooKeeper's uses ephemeral nodes which exist as long as there is a client session, and get deleted as soon as the session ends. As you may have guessed, the first service instance that starts up initiates a session in the ZooKeeper server and creates an ephemeral node to become the master. All

other service instances from the same cluster become slaves and start waiting for the ephemeral node to be released. This is how this a ZK based implementation makes sure there is only one active service instance in the whole cluster, ensuring a master/slave failover behaviour.
Rather the managing a separate ZooKeeper cluster, an even better option would be to use the Etcd cluster tha ships within Kubernetes. Etcd provides the necessary building blocks for implementing leader election functionality, and there are few client libraries that have implemented the functionality already. An implementation with Etcd (or any other distributed lock implementation) would be similar to the one with ZooKeeper where only one instace of the service becomes master/leader and becomes active, and other service instances are passive and wait for the lock. This will ensure that even if there are multiple pods with the same service, and they are all health, and may be certain functionality of the services in the pod are function, some other part is singleton and active on one random pod instance.

Pod Disruption Budget

While singleton service and leader election are trying to limit the maximum number of instances a service is running at a time, the Pod Disruption Budget functionality provides the opposite functionality and it is limiting the maximum number of instances that are running at a time. At its core, it ensures a certain number or percentage of pods will not voluntarily be evicted from a node at any one point in time. Voluntary here means an eviction that can be delayed for a certain time, for example when it is triggered by draining a node for maintenance or upgrade (kubectl drain), and cluster autoscaling down.

```
apiVersion: policy/v1alpha1
kind: PodDisruptionBudget
metadata:
  name: disruptme
spec:
  selector:
    matchLabels:
      name: myapp5pods
  minAvailable: 80%
```

This functioanily is useful with quorum-based applications that require a minimum number of replicas running at all times to ensure a quorum. Or maybe some part of your application serving the load never goes below certain percentage of the total. Even if not directly related to a singleton behaviour, Pod Disruption Budget feature also controls the number of service instances that run at a time and worth mentioning in this chapter.

More Information

Simple leader election with Kubernetes and Docker
Camel master component
Leader election in go client by Kubernetes
Configuring a Pod Disruption Budget by Kubernetes

11. Self Awareness

There are many occasions where an application needs to be self-ware and have information about itself and the environment where it is running. Kubernetes provides simple mechanisms for introspection and metadata access and also APIs for more complex information querying.

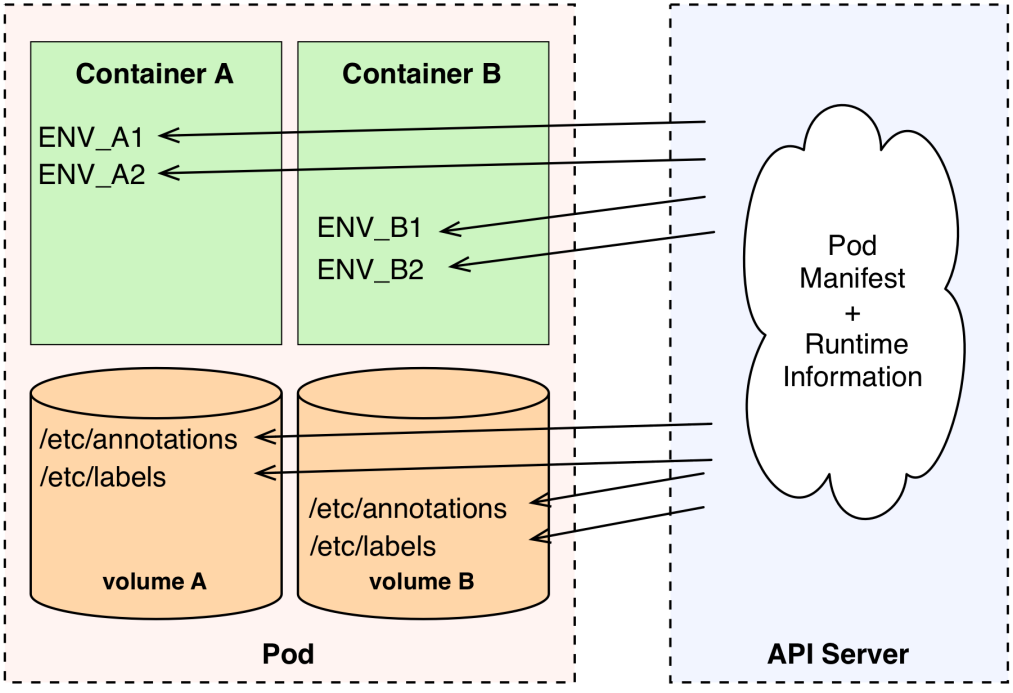
Problem

The majority of cloud native applications are disposable, stateless, applications without an identity, treated as cattle rather than pets. But even with this kind of applications you may want to know information about the application itself and the environment it is running in. That may include information that is known only at runtime such as the pod name, pod IP, the host name on which the application is running. Or other static information that is defined at pod level such as the pod resource requests and limits specified, or some dynamic information such as annotations and labels that may be altered by the user at any moment. Such information is required in many different scenarios, for example, depending on the resources made available to the container, you may want to tune the application thread pool size, or memory consumption algorithm. You may want to use the pod name and the host name while logging information, or while sending metrics to a centralized location. You may want to discover other pods in the same namespace with a specific label and join them into a clustered application, etc. For this and other similar cases, Kubernetes offers few ways for containers to introspect themselves and retrieve useful metadata.

Solution

The challenges described above, and the following solution are not specific only to containers, but exists in many dynamic environments. For example AWS offers

Instance Metadata and User Data services that can be queried from any EC2 instance to retrieve metadata about EC2 instance itself. Similarly AWS ECS provides APIs that can be queried by the containers and retrieve information about the container cluster. The Kubernetes approach is even more elegant and easier to use. The so-called Downward API allows passing metadata about the pod and the cluster through environment variables and volumes. These are the same familiar mechanisms we used for passing application related data through ConfigMap and Secret. But in this case, the data is not defined by us, instead we specify the keys, and the values are coming from Kubernetes.



Application introspection

The main point from the above diagram is that the metadata is injected into your pod and made available locally. The application does not need to call a remote API to retrieve the data. Let's see how easy is to request metadata through environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: introspect
spec:
  containers:
  - name: main
    image: busybox
    command: ["sleep", "9999999"]
    env:
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
```

Certain metadata such as labels and annotations can be changed by the user while the pod is running. And using environment variables cannot reflect such a change unless the pod is restarted. For that reason this metadata is not available as environment variables but available through volumes only.

```
apiVersion: v1
kind: Pod
metadata:
  name: introspect
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
```

```
spec:
   containers:
    - name: main
      image: busybox
      command: ["sleep", "9999999"]
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

With volumes, if the metadata changes while the pod is running, it will be reflected into the volume files. Then it is up to do consuming application to detect the file change and use the updated data accordingly.

Accessing the Kubernetes API

One of the downsides of Downward API is that it offers a small fixed number of keys that can be referenced. If your application needs more data about itself, or any other kind of cluster related metadata that is available through the API server, it can query the API server directly. This technique is used by many applications that query the API server to discover other pods in the same namespace that have certain labels or annotations. Then the application may form a some kind of cluster with the discovered pods and syc state. It is also used by monitoring applications to discover pods of interest and then start instrumenting them. In order to interact with the Kubernetes API server, there are many client libraries available for different

languages. You will also need information such as account token, certificate and the current namespace to use as part of your query which is all provided in every container at the following location: /var/run/secrets/kubernetes.io/serviceaccount/.

More Information

Using a DownwardApiVolumeFile by Kubernetes
Instance Metadata and User Data by Amazon
Amazon ECS Container Agent Introspection by Amazon

III Structural Patterns

The patterns in this category are focused on the relationships among containers and oraganizing them appropirately for the different use cases.

One way to think about container images and containers is similar to classes and objects in the object oriented world. Container images are the blueprint from which containers are instantiated. But these containers do not run in isolation, they run in another abstraction called pod that provide unique runtime capabilities. Containers in a pod are independent is some aspects with independent lifecycle, but share common faith in some other aspects. All these forces among containers in a pod give birth to new structural patterns which we will look at next.

12. Sidecar

A sidecar container extends and enhances the functionality of a preexisting container without changing it. This is one of the fundamental container patterns that allows single purpose build containers to cooperate closely together for a greater outcome.

Problem

Containers are a popular packaging technology that allows developers and system administrators to build, ship and run applications. A container represent a natural boundary for a unit of functionality that has its distinct runtime, release cycle, API and owning team. A good container, behaves like a single linux process, solves one problem and does it well, and it is created with the idea of replaceability and reuse. The last part is very important as it allows us to build applications more quickly, by leveraging existing specialized containers. The same way as we do not have to write any longer a Java library for an HTTP client but simply use one of the existing ones, we do not have to create a container for a web server, but use one of the existing ones. This will let the teams avoid reinventing the wheel and creates an ecosystem with smaller number of better quality containers to maintain. But having single purpose specialized and reusable containers, requires a way of collaboration among containers to provide the desired collective outcome. The Sidecar pattern describe this kind of collaboration where a container extends and enhances the functionality of another preexisting container.

Solution

In Automatable Unit chapter, we saw how the pod primitive allows us to combine multiple containers into a single deployment unit. The pod is such a fundamental primitive that it is present in many cloud native platforms under different names, but providing similar capabilities. A pod as the deployment unit, puts certain runtime

constraints on the containers belonging to it. For example all containers end up deployed to the same host and they do share the same pod lifecycle. But at the same time, pods allow sharing volumes, communication over the local network or host IPC, etc to all its containers. This characteristics gives different reasons for users to put a group of containers into a pod. Sidecar (in some places also referenced as Sidekick) is used to describe the scenario where a container is put into a pod to extend and enhance another container's behaviour.

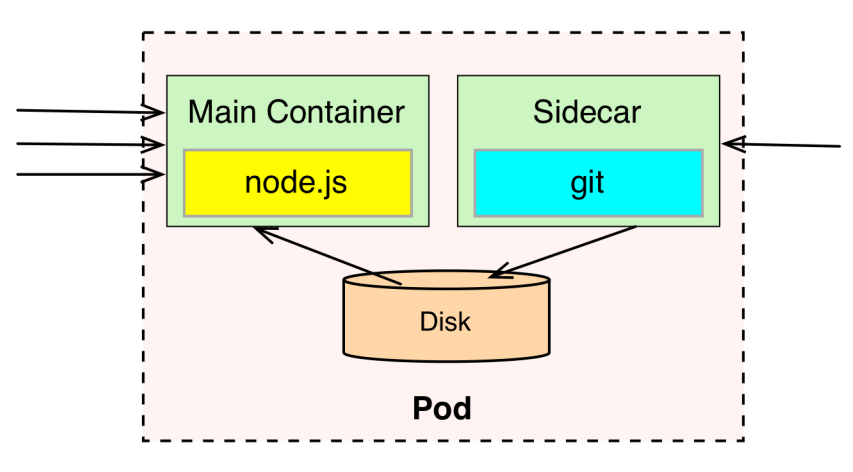
Examples

The most common example used to demonstrate this pattern is with a HTTP server and the git synchronizer. The HTTP server container is only focused on serving files and has no knowledge of how and where the files are coming from. Similarly, the git synchronizer container's only goal is to sync files from a git server to the local file system. It does not care what happens to the files once synced, and its only concern is about keeping the folder in sync with the git server. Here is an example pod definition with these two containers configured to use a volume for file exchange.

```
apiVersion: v1
kind: Pod
metadata:
  name: init
  labels:
    app: init
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "download",
            "image": "axeclbr/git",
            "command": [
                "git",
                "clone",
                "https://github.com/mdn/beginner-html-site-scripted",
                "/var/lib/data"
            ],
```

```
            "volumeMounts": [
                {
                    "mountPath": "/var/lib/data",
                    "name": "git"
                }
            ]
        }
    ]'
spec:
  containers:
  - name: run
    image: docker.io/centos/httpd
    ports:
      - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html
      name: git
  volumes:
  - emptyDir: {}
    name: git
```

This an example where the git synchronizer enhances the HTTP server's behaviour with content to serve. We could also say that both containers collaborate and they are equally important in this use case, but typically in a Sidecar pattern there is the notion of main container and the helper container that enhances the common behaviour.



Sidecar Pattern

This simple pattern allows runtime collaboration of containers, and at the same time enables separation of concerns for both containers, which might be owned by separate teams with different release cycles, etc. It also promotes replaceability and reuse of containers as node.js and git synchronizer can be reused in other applications and different configuration either as a single container in a pod, or again in collaboration with other containers.

More Information

Design patterns for container-based distributed systems by Brendan Burns and David Oppenheimer
Prana: A Sidecar for your Netflix PaaS based Applications and Services by Netflix

13. Initializer

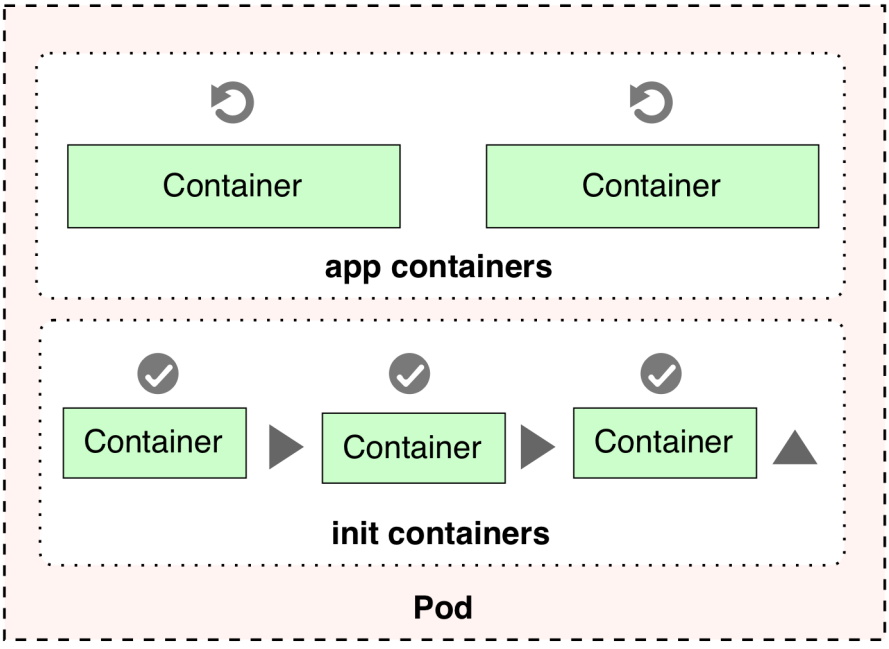
Init containers in Kubernetes allow separation of initialization related tasks from the main application logic. This separation ensures all initialization steps are completed in the defined sequence successfully before starting the main application containers. With this feature at hand, we can create containers that are focused on single initialization tasks and main application focused tasks.

Problem

Initialization is a very common concern in many programing languages. Some languages have it covered as part of the language, and some use naming conventions and patterns to indicate a construct as the initializer. For example, in the Java programing language, to instantiate an object that requires some setup, we use the constructor (or static blocks for more fancy use cases). Constructors are guaranteed to run as the first thing withing the object, and they are guaranteed to run only once by the managing runtime (this is just as an example, let's not go into details of different languages and corner cases). And we can use the constructor to validate the pre-conditions for the existence of an object, which are typically mandatory parameters, or to initialize the instance fields with incoming arguments or default values. The idea of container initializers is a similar one, but at container level rather than class level. So if you have one or more containers in a pod that represent your main application, these containers may have prerequisites before starting up. That may include setting up special permissions on the file system, schema setup in a database, application seed data that has to be installed, etc. On the other hand these initializing logic may require tools and libraries that are not included in the application image. Or for security reason, the application image may not be allowed to perform the initializing activities itself. Or you may want to delay the startup of your application until an external dependency is satisfied. For all this kind of use cases, there is the concept of init containers which allows separation of initializing activities from the main application activities, both at development and runtime.

Solution

Init containers in Kubernetes are part of the pod definition, and they separate the containers in a pod in two groups: init containers and application containers. All init containers are executed in a sequence and all of them have to terminate successfully before the application containers are started. In a sense, the init containers are like construction instructions in a class that helps for the object initialization.



Init and app containers in a pod

For most of the cases, init containers are expected to be small, run quickly and complete successfully. Except of the use case where an init container is used to delay the start of the pod while waiting for a dependency. If an init container fails, the whole pod is restarted again (unless it is marked with RestartNever) causing also other init containers to run again. Thus making init containers idempotent is mandatory.

One could ask: why separate containers in a pod in two groups, why not just use any

container in a pod for initialization if required? The answer is that these two groups of containers have different lifecycle, purpose and even authors in some cases. From one hand, all the containers are part of the same pod, so they do share resource limits, volumes, security settings and end up placed on the same host. Init containers have all of the same capabilities of application containers, but they also have slightly different resource handling and health checking semantics. For example there is no readiness check for an init container as all init containers must terminate successfully before a pod startup can continue with application containers. Init containers also affect the way in which pod resource requirements are calculated for scheduling, autoscaling and quotas. Given the ordering in the execution of all containers in a pod (first init containers run a sequence, then all application containers run in parallel), the effective pod level request/limit values become the highest values of the following two groups:

- the highest init container request/limit value; or
- the sum of all application container values;

The consequence of this behaviour is that you may have an init container that requires a lot of resources but runs only for few seconds before the main application container starts up which may require much less resources. Regardless of that, the pod level request/limit values affecting scheduling will be based on the higher value of the init container which is not the best usage of resources.
And last but not least, init containers allow keeping containers single purposed. An application container can focus on the application logic only and keep it single purposed configurable container created by the application engineer. An init container, can be authored by a deployment engineer and focus on configuring an application specific for the use case. We demonstrate this in the following example where we have one application container based on httpd which serves files. The container provides a generic httpd capability and does not make any assumptions where the files to serve might come for the different use cases. In the same pod, there is also an init container which provides git client capability and its sole purpose is to clone a git repo. And since both containers are part of the same pod, they can access the same volume to share data. This same mechanism has been used to share the cloned files from the init container to the main application container.

```
apiVersion: v1
kind: Pod
metadata:
  name: init
  labels:
    app: init
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "download",
            "image": "axeclbr/git",
            "command": [
                "git",
                "clone",
                "https://github.com/mdn/beginner-html-site-scripted",
                "/var/lib/data"
            ],
            "volumeMounts": [
                {
                    "mountPath": "/var/lib/data",
                    "name": "git"
                }
            ]
        }
    ]'
spec:
  containers:
  - name: run
    image: docker.io/centos/httpd
    ports:
      - containerPort: 80
    volumeMounts:
    - mountPath: /var/www/html
      name: git
  volumes:
```

```
  - emptyDir: {}
    name: git
```

This is a very simple example to demonstrate how init containers works. We could have achieved the same effect using ConfigMap, or PersistentVolumes that is backed by git for example. Or using the Sidecar pattern where the httpd container run and the git container run side by side. But init containers offer more flexibility with ability to run multiple containers until completion before starting the application containers, which is required for many complex application setup scenarios.

More Information

Init Containers by Kubernetes
Configuring Pod Initialization by Kubernetes
The Initializer Pattern in JavaScript
Object Initialization in Swift

14. Ambassador

This pattern provides a unified view of the world to your container. This is a specialization of the Sidecar pattern, where in this situation, the Ambassador does not enhance the behaviour of the main container. Instead, it is used with the only purpose of hiding complexity and providing a unified interface to services outside of the pod.

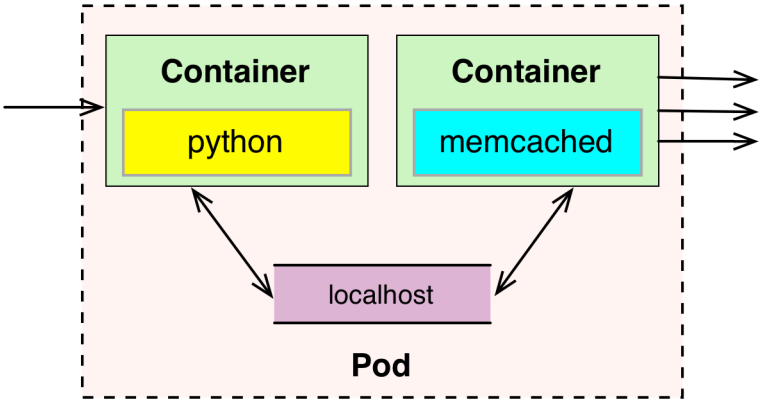
Problem

There are many occasions where a container has to consume a service that is not easy to access. The difficulty in the accessing the service may be due to many reasons such as dynamic and changing address, the need for load balancing of clustered service instances, etc.
Considering that containers ideally should be single purposed and reusable in different contexts, we may have a container that performs some kind of processing by consuming an external service. But consuming that external service may require some special service discovery library that we do not want to mix with our container. We want our container to provide its processing logic by consuming different kind of services, using different kind of service discovery libraries and methods. This technique of isolating the logic for accessing services in the outside world is described by the Ambassador pattern.

Solution

Let's see few examples here to demonstrate the pattern.
The very first example that comes into mind is using a cache in an application. Accessing a local cache for the development environment may be a simple configuration, but on the production environment we may need a client configuration the is able to connect to the different shards of the cache.

Another example would be consuming a service that requires some kind of service discovery logic on the client side.
A third example would be consuming a service over a non-reliable protocol such as HTTP where we have to perform retry, use circuit breaker, configure time outs, etc.
In all these cases, we can use an Ambassador container that hides the complexity of accessing the external services and provides a simplified view and access to our main application container over `localhost`.



Ambassador Pattern

The slight difference of this pattern to Sidecar is that, an Ambassador does not enhance the main application with additional capability, instead it acts merely as a smart proxy to the outside world, where it gets its name from. And having a different name than the generic cooperating Sidecar pattern communicates the purpose of the pattern more precisely.
The benefits of this pattern are similar to those of Sidecar pattern where it allows keeping containers single purposed and reusable. With such a pattern, our application container can focus on performing its processing and delegate the responsibility and specifics of consuming the external service to another container. This also allows us creating of specialized and reusable Ambassador containers that can be combined with other application containers.

More Information

How To Use the Ambassador Pattern to Dynamically Configure Services on CoreOS
Dynamic Docker links with an ambassador powered by etcd by Docker
Link via an ambassador container by Docker
Modifications to the CoreOS Ambassador Pattern by Christian Zunker

15. Adapter

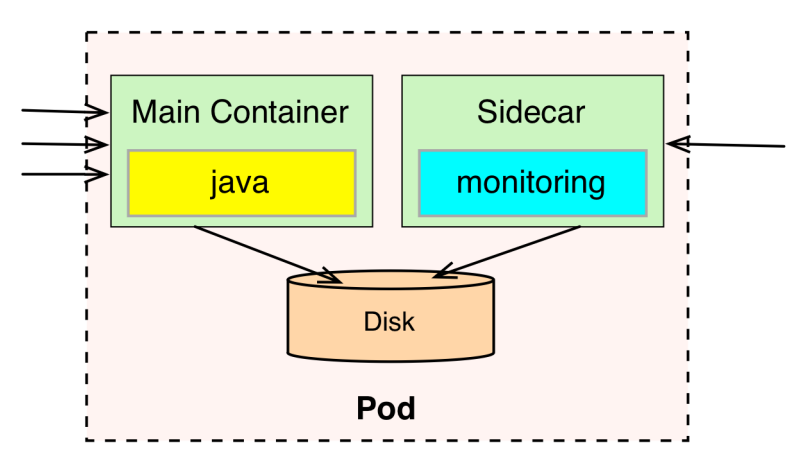
An Adapter is kind of reverse Ambassador and provides a unified interface to a pod from the outside world.

Problem

We will discuss the context and the solution in the next section.

Solution

This is another variation of the Sidecar pattern and the best way to illustrate it is through an example. A major requirement for distributed applications is detailed monitoring and alerting. And using containers enables us to use different languages and libraries for implementing the different components of the solution. This creates a challenge for monitoring such an heterogeneous applications from a single monitoring solution which expects a unified view of the system. To solve this, we could use the Adapter pattern to provide a unified monitoring interface by exporting metrics from various application containers (whether that is language or library specific) into one common format.



Adapter Pattern

Another example would be logging. Different containers may log information in different format and level of details. An Adapter can normalize that, clean it up, enrich with some context and then make it available for scraping by the centralized log aggregator. In summary, this pattern takes an heterogeneous system and makes it conform to a consistent unified interface with standardise and normalized format that can be consumed by the outside world.

More Information

Container patterns for modular distributed system design by Brendan Burns - VIDEO

IV Configuration Patterns

Every non-trivial application needs to be configured somehow. The easiest way to do so is to store configuration information directly into the source code. This has also the nice side effect that the configuration and code live and dies together as it can't be changed from the outside, so it's perfectly suited for the immutable server pattern, isn't it ? Well, as much as we agree with this paradigm to create an image once and never change it again, we still need the flexibility to adapt configuration without recreating the application image every time. In fact this would be not only time and resource consuming but it is also an anti-pattern for a Continuous Delivery approach, where the application is created once and then *moved unaltered* through the various stages of the CD pipelines until the application will end up finally in production. In such a scenario how would one then adapt to the different setups for development, integration and production environments then ? The answer is to use *external configuration*, referred to by the application and which is different for each environment.

The patterns in this category are all about how applications can be customised by external configuration and how to adapt to the various runtime environments:

- *EnvVar Configuration* uses environments variables to store configuration data as the twelve-factor app manifesto demands.
- *Configuration Resource* uses Kubernetes resources like config-maps or secrets to store configuration information.
- *Configuration Template* is useful when large configuration files needs to be managed for various environments which differ only slightly.
- *Immutable Configuration* brings immutability to large configuration sets by putting it into containers which are linked to the application during runtime.
- *Configuration Service* is a dedicated lookup service for configuration data which can easily be updated during runtime.

Links

- Docker Configuration Pattern explains how plain Docker can deal with external configuration.

16. EnvVar Configuration

For small sets of configuration values, the easiest way to externalise the configuration is by putting them into environment variables which are directly supported by every runtime platform.

Problem

Every non-trivial application needs some configuration for accessing databases or external web services. Not only since the Twelve-Factor App manifesto we know that it is a bad thing to hard code this configuration within the application. Instead, the configuration should be *externalised* so that we can change it even after the application has been built.

But how can this be done best in a containerised world?

Solution

The Twelve-Factor App Manifesto recommends using **environment variables** for storing application configuration. This approach is simple and works for any environment and platform. Every operating system knows how to define environment variables and how to propagate them to applications and every programming language also allows easy access to these environment variables. It is fair to claim that environment variables are universally applicable. When using environment variables, a common use pattern is to define hard coded default values during build time which we then can overwrite at run time.

Let's see with some concrete examples how this works in a Docker and Kubernetes world.

Example

For Docker images, environment variables can be defined directly in Dockerfiles with the `ENV` directive. You can define them line-by-line but also all in a single line:

```
FROM jboss/base-jdk:8
ENV DB_HOST "dev-database.dev.intranet"
ENV DB_USER "db-develop"
ENV DB_PASS "s3cr3t"

# Alternatively:
ENV DB_HOST=dev-database.dev.intranet DB_USER=db-develop DB_PASS=s3c\
r3t
...
```

A Java application running in such a container then can easily access the variables with a call to the Java standard library:

```
public String getMongoDbConnect() {
  return String.format("mongodb://%s:%s@%s",
                       System.getenv("DB_USER"),
                       System.getenv("DB_PASS"),
                       System.getenv("DB_HOST"));
}
```

Directly running such an image will try to connect to your default database that you have defined. As explained above in most cases you want to override these parameters from outside of the image, though.

When running such an image directly with Docker then environment variables can be used from the command line:

```
docker run -e DB_HOST="prod-database.prod.intranet" \
           -e DB_USER="db-prod" \
           -e DB_PASS="3v3nm0r3s3cr3t" \
           acme/bookmark-service:1.0.4
```

For Kubernetes, this kind of environment variables can be set directly in a pod specification of a controller like `Deployment` or `ReplicaSet`:

```
apiVersion: v1
kind: ReplicaSet
spec:
  replicas: 1
  template:
    spec:
      containers:
      - env:
        - name: DB_HOST
          value: "prod-database.prod.intranet"
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: "db-passwords"
              key: "monogdb.password"
        - name: DB_USER
            valueFrom:
              configMapKeyRef:
                name: "db-users"
                key: "mongodb.user"
        image: acme/bookmark-service:1.0.4
```

In such a pod template you can not only attach values directly to environment variables (like for `DB_HOST`), but you can also use a delegation to Kubernetes `Secrets` (for confidential data) and `ConfigMaps` (for typical configuration). `Secret` and `ConfigMap` are explained in detail in the pattern *Configuration Resource* The big advantage of this indirection is that the value of the environment variables can be managed independently from the Pod definition.

About default values

Default values make life easier as they take away the burden to select a value for a configuration parameter you might not even know that it exists. They also play a significant role in a *convention over configuration* paradigma. However defaults are not always a good idea. Sometimes they might be even an anti-pattern for an evolving application. This is because *changing* default values in retrospective is a difficult task in general. First, changing default values means to change them within the code which requires a rebuild. Second, people relying on defaults (either by convention or consciously) will always be surprised when a default value changes. We have to communicate the change, and the user of such an application has probably to modify the calling code as well. Changes in default values, however, might often make sense, also because it is so damned hard to get default values right from the very beginning. It's important that we have to take a change in default values as to a *major change* and if semantic versioning is in use, such a modification justifies a bump in the major version number. If unsatisfied with a present default value it is often better to remove the default altogether and throw an error if the user does not provide a configuration value. This will at least break the application early and prominently instead of doing something different and unexpected silently. Considering all these issues it is often the best solution to *avoid default values* from the very beginning if you can not be 90% sure that there is a good default which will last for a long time. Database connection parameters, even though used in our example, are good candidates for not providing default values as they highly depend on the environment and can often not be reliably predicted. Also if we do not use default values, then the configuration information which has to be provided now explicitly can serve nicely as documentation, too.

Discussion

Environments variables are super easy to use. Everybody knows about environment variables and how to set them. This operation system concept maps smoothly to containers, and every runtime platform supports environment variables. Environment variables are ideal for a decent amount of configuration values. However, when there

is a lot of different parameters to configure, the management of all these environment variables becomes unwieldily. In that case, many people use an extra level of indirection where we put configuration into various configuration files, one for each environment. Then a single environment variable is used to select one of these files. Profiles from Spring Boot are an example of this approach. Since these profile configuration files are typically stored in the application itself within the container, it couples the configuration tightly to the application itself. Often configuration for development and production then ends up side by side in the same Docker image, which requires an image rebuild for every change in either environment. *Configuration Template*, *Immutable Configuration* and *Configuration Resource* are good alternatives when more complex configuration needs to be managed.

Because environment variables are so universally applicable, we can set them at various levels. This leads to fragmentation of the configuration definition so that it ‘s hard to track for a given environment variable where it comes from. When there is no central place where all environments variables are defined, then it is hard to debug configuration issues. Another disadvantage of environment variables is that they can be set only **before** an application start and we can not change it afterwards. On the one hand, this is a drawback that you can’t change configuration “hot” during runtime to tune the application. However many see this also as an advantage as it promotes *immutability* also to the configuration. Immutability here means that you throw away the running application container and start a new copy with a modified configuration, very likely with some smooth deployment strategy like rolling updates. That way you are always in a defined and well known configurational state.

More Information

- The Twelve-Factor App Manifesto
- Immutable Server Pattern
- Spring Boot profiles for using sets of configuration values

17. Configuration Resource

For complex configuration data Kubernetes provides dedicated configuration resource objects which can be used for open and confidential data, respectively.

Problem

One of the big disadvantages of the *EnvVar Configuration* pattern is that it's only suitable for up to a handful of variables. Also because there are various places where environment variables can be defined, it is often hard to find the definition of a variable. And even if you find it you can't be entirely sure whether it is not overridden in another spot. For example, environment variables defined within a Docker image can be replaced during runtime in a Kubernetes `Deployment` configuration. Often it also makes much sense to keep all the configuration data at a single place and not scattered around in various resource definition files. So some extra indirection would help to allow for more flexibility. Another limitation of the environment variable approach is that it is suitable only when the amount of configuration is small. It does not make any sense at all to put the content of a whole configuration file into an environment variable.

Solution

Kubernetes provides dedicated configuration resources which are more flexible than pure environment variables. We can use the objects `ConfigMaps` and `Secrets` for plain and confidential data, respectively.

We can use both in the same way as they are storage for simple key-value pairs. In the following when referring to `ConfigMaps`, the same can be applied mostly to `Secrets`, too. The section ConfigMap versus Secrets describes the differences, but we will explain it also on the way.

We can use the keys of a config-map in two different ways:

- as a reference for *environment variables*, where the key is the name of the environment variable.
- as *files* which are mapped to a volume mounted in a pod. The key is used as the file name.

The file in a mounted config-map volume is updated when the `ConfigMap` is updated via the Kubernetes API. So, if an application supports hot reload of configuration files, it can immediately benefit from such an update. For `Secrets` it is different, though. When we use secrets in volumes, they don't get updated during the lifetime of a pod, which means that we cannot update secrets on the fly. The same goes for config-map entries which are used as environment variables since environment variables can't be changed after a process has been started.

A third alternative is to store configuration directly in external volumes which are then mounted. Volumes of type `gitRepo` are best suited for storing configuration data. We explain the git backed volumes with some example below.

Example

The following examples concentrate on `ConfigMap` usage, but the same can be easily used for `Secrets`, too. There is one big difference, though: Values for secrets have to be Base64 encoded.

A `ConfigMap` is a resource object which contains key-value pairs in its `data` section:

```
apiVersion: v1
  kind: ConfigMap
  metadata:
    name: spring-boot-config
  data:
    JAVA_OPTIONS: "-Djava.security.egd=file:/dev/urandom"
    application.properties: |
      # spring application properties file
      welcome.message=Hello from a Kubernetes ConfigMap!!!
      server.port=8080
```

We see here that a config-map can also carry the content of complete configuration files, like a Spring Boot `application.properties` in this example. You can imagine that for non-trivial use case this section can get quite large!

Instead of manually creating the full resource descriptor, we can use `kubectl', too to create config-maps or secrets. For the example, the equivalent` kubectl‘ command looks like

```
kubectl create cm spring-boot-config \
   --from-literal=JAVA_OPTIONS=-Djava.security.egd=file:/dev/urandom\
 \
   --from-file=application.properties
```

This config-map then can be used in various places. I.e. a config-map entry can be used everywhere where environment variables are defined:

```
apiVersion: v1
kind: ReplicaSet
spec:
  replicas: 1
  template:
    spec:
      containers:
      - env:
        - name: JAVA_OPTIONS
          valueFrom:
            configMapKeyRef:
              name: spring-boot-config
              key: JAVA_OPTIONS
....
```

For using a secret instead replace `configMapKeyRef` with `secretKeyRef`.

When used as a volume, the complete config-map is projected into this volume, where the keys are used as file names:

```
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-app
spec:
  containers:
    - name: web
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: spring-boot-config
```

This configuration results in two files in `/etc/config': An` application.properties `with the content defined in the config map, and a` JAVA_OPTIONS `file with a single line content. The mapping of config data can be tuned fine granularly by adding additional properties to the volume declaration. Please refer to the` ConfigMap‘ documentation for more details.

ConfigMap versus Secrets

As we have seen, config-maps and secrets are quite similar. There are some subtle differences, though:

- Values of secrets are always Base64 encoded and can be binaries. In contrast, config-map values are always plain text.
- Secrets that are mounted as volumes are *not* refreshed when the value within the secret changes. Mounted volumes of config-maps have their content updated when config-map changes.

Configuration from Git repositories

Another possibility for storing configuration with the help of Kubernetes resources is the usage of `gitRepo` volumes. It mounts an empty directory on your pod and

clones a git repository into it. Thie mount happens during startup of the pod, and the local cloned repo won't be updated automatically. In that sense, it is similar to the approach described in the previous chapter by using init containers with templates. The difference is that `gitRepo` volumes require a potentially external access to a Git repository which is not a Kubernetes resource. This external dependency needs to be monitored separately. The advantage of a `gitRepo` based configuration is that you get versioning and auditing for free. Another example shows how we can use `gitRepo` volumes in practice.

Discussion

The biggest advantage of using config maps and secrets is that they decouple the *definition* of configuration data from its *usage*. This decoupling means that we can manage the objects which use config-maps independently.

Another benefit of these objects is that they are intrinsic features of the platform. No individual constructs like in the *Immutable Configuration* pattern is required. Also, config maps and secrets allow the storage of configuration information in dedicated resource objects which are easy to find over the Kubernetes API.

However, config maps and secrets also have their restrictions: With 1 MB they can't store arbitrarily large data and are not well suited for non-configurational application data. You can also store binary data in config-maps, but since they have to be Base64 encoded you can use only around 700kb data for it.

Real world Kubernetes and OpenShift clusters also put an individual quota on the number of config maps which can be used per namespace or project, so config-map are not golden hammers.

In the next chapter, we see now how we can come over those size limitations by using *Configuration Templates*

More Information

- Kubernetes Documentation for ConfigMap, Secrets and `gitRepo` Volumes
- Size restriction of `ConfigMap`

18. Configuration Template

For complex and large configuraton data use templates which are processed during application startup to create the real configuration from environment specific values.

Problem

In the previous chapter on Configuration Resource we have seen how we can use the Kubernetes native resource object `ConfigMap` and `Secret` for configuring our applications. But we all know how large configuration files can get. Putting them directly into config-maps can be problematic since they have to be properly embedded in the resource definition. It is easy to break the Kubernetes resource syntax, and you need to be careful to escape special characters like quotes. The size limit for the sum of all values of config-maps or secrets is 1 MB (which is the limit of the underlying backend store `etcd`).

Also, different execution environments often have a similar configuration which differs only slightly. This similarity leads to a lot of redundant settings in the config-maps because it is mostly the same in each environment.

Solution

To reduce the duplication, it makes sense to only store the *differing* configuration values like database connection parameters in a config-map or even directly in environment variables. During startup of the container, these values are processed with configuration templates to create the full configuration file (like a Wildly `standalone.xml`). There are many tools like Tillerï¿¼ (Ruby) or gomplate (Go) for processing templates during runtime in a flexible way during application initialization. Here a configuration template is filled with data coming from environment variables or a mounted volume, possibly backed by a config-map.

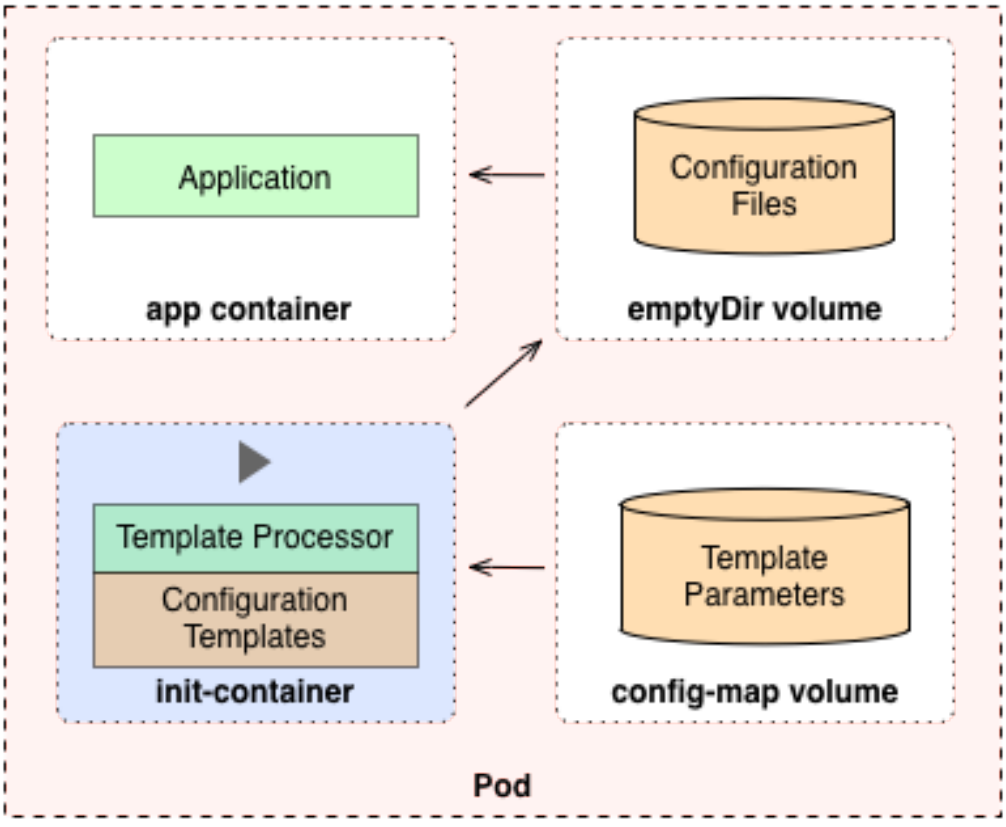
The fully processed configuration file is put into place before the application is started so that it can be directly used like any other configuration file.

There are two techniques how such a live processing can happen during runtime:

- We can add the template processor as part of the `ENTRYPOINT` to a Dockerfile and so the template processing becomes directly part of the Docker image. The entry point here is typically a script which first performs the template processing and then starts the application. The parameters for the template come from environment variables.
- For Kubernetes the perfect place where to perform initialisation is with an init-container of a pod in which the template processor runs an creates the configuration for the 'real' containers in the pod. init-containers are described in details in the Initializer pattern.

For Kubernetes the init-container approach is the most appealing because we can use config-maps directly for the template parameters.

The following diagram illustrates how this pattern.



Configuration Template Pattern

The application's pod definition consists of at least two containers: One init-container for the template processing and the application container. The init-container contains not only the template processor but also the configuration templates themselves. In addition to the containers, this pod also defines two volumes: One volume for the template parameters, backed by a config-map. The other volume is an `emptyDir` volume which is used to share the processed templates between the init-container and the application.

With this setup, the following steps are performed during startup of this pod:

- The init-container is started and runs the template processor. The processor takes the templates from its image, the template parameters from the mount

config-map volume and stores the result in the `emptyDir` volume.
- After the init-container has finished, the application container starts up and loads the configuration files from the `emptyDir` volume.

Example

The following example uses an init-container for managing a full set of Wildfly configuration files for two environments: A development environment and a production environment. Both are very similar to each other and differ only slightly. In fact, in our example they differ only in the way how logging is performed: Each log line is prefixed with `DEVELOPMENT:` or `PRODUCTION:`, respectively.

You can find the full example along with full installation instructions in our examples GitHub repo. We are showing only the concept here, for the technical details, please refer to the source repo.

The log pattern is stored in `standalone.xml` which we parameterise by using the Go Template language:

```
....
<formatter name="COLOR-PATTERN">
  <pattern-formatter pattern="{{(datasource "config").logFormat}}"/>
</formatter>
....
```

We use gomplate as our template processor here. `gomplate` uses the notion of a *datasource* for referencing the template parameters to be filled in. In our case, this data source comes from a config-map backed volume which is mounted to an init-container.

Here, the config-map contains a single entry with the key `logFormat` from where the actual format is extracted.

With this template in place, we can now create the Docker image for the init-container. The Dockerfile for the image "k8spatterns/example-configuration-template-init" is very simple:

```
FROM k8spatterns/gomplate
COPY in /in
```

The base image k8spatterns/gomplate contains the template processor and an entry point script which uses the following directories by default:

- `/in` holds the Wildfly configuration templates, including the parameterised `standalone.xml`, including the parameterised `standalone.xml`. These are added directly to the image.
- `/params` is used to lookup the `gomplate` data sources, which are YAML files. This directory is mounted from a config-map backed pod volume.
- `/out` is the directory into wich the processed files are stored. This directory is mounted in the Wildfly application container and used for the configuration.

The second ingredient of our example is the config-map holding the parameters. We use a simple file

```
logFormat: "DEVELOPMENT: %-5p %s%e%n"
```

A config-map `wildfly-parameters` will contain this YAML formatted data referenced by a key 'config.yml' and picked up by an init-container.

Finally, we need the `Deployment` resource for the Wildlfy server:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    example: cm-template
  name: wildfly-cm-template
spec:
  replicas: 1
  template:
    metadata:
      labels:
```

```
      example: cm-template
  spec:
    initContainers:
    - image: k8spatterns/example-config-cm-template-init
      name: init
      volumeMounts:
      - mountPath: "/params"
        name: wildfly-parameters
      - mountPath: "/out"
        name: wildfly-config
    containers:
    - image: jboss/wildfly:10.1.0.Final
      name: server
      command:
      - "/opt/jboss/wildfly/bin/standalone.sh"
      - "-Djboss.server.config.dir=/config"
      ports:
      - containerPort: 8080
        name: http
        protocol: TCP
      volumeMounts:
     - mountPath: "/config"
        name: wildfly-config
    volumes:
    - name: wildfly-parameters
      configMap:
        name: wildfly-parameters
    - name: wildfly-config
      emptyDir: {}
```

This declaration is quite a mouthful, so let's drill it down: This deployment specification contains a pod with our init-container, the application container, and two internal pod volumes:

- The first volume "wildfly-parameters" contains our config-map with the same name (i.e. it contains a file called `config.yml` holding out parameter value).

- The other volume is an empty directory initially and is shared between the init-container and the Wildfly container.

If you start this deployment the following steps will happen:

- An init-container is created and its command is executed. This container takes the `config.yml` from the config-map volume, fills in the templates from the `/in` directory in an init container and stores the processed files in the `/out` directory. The `/out` directory is where the volume `wildfly-config` is mounted.
- After the init-container is done, a Wildfly 10 server starts with an option so that it looks up the complete configuration from the `/config` directory. Again, `/config` is the shared volume `wildfly-config` containing the processed template files.

It is important to note that we do **not** have to change these deployment resource descriptor when going from the *development* to the *production* environment. Only the config map with the template parameters is different.

With this technique, it is easy to create DRY configuration without copying and maintaining duplicated large configuration files. Eg. when the Wildfly configuration changes for all environments, only a single template file in the init container needs to be updated. This approach has, of course, significant advantages on maintenance as the there is not even the danger for the configurations files to diverge.

> A #### Volume debugging tip A A When working with pods and volumes like in this patterns it is not obvious how to debug if things don't work as expected. A Here are two tips how you can have a look at the mounted volumes A A * Within the pod the directory `/var/lib/kubelet/pods/{podid}/volumes/kubernetes.io~empty-dir/` contains the content of an `emptyDir` volume. Just `kubectl exec` into the pod when it is running, examining this directory A * For debugging the outcome of init-containers it helps if the *command* of the primary container is replaced temporarily with a dummy sleep command so that you has time to examine the situation. This trick makes especially sense

if you init-container fails to start-up and so your application fails to start because the configuration is missing or broken. The following command within the pod declaration gives you an hour time to debug the volumes mounted, again with `kubectl exec -it <pod> sh` into the pod: A A `A command: A - /bin/sh A - "-c" A - "sleep 3600" A`

Discussion

Configuration Template builds on top of *Configuration Resource* and is especially suited for situations where we need to operate applications in different environments. These applications often require a considerable amount of configuration data from which only a small fraction is dependent on the environment. Even when copying over the whole configuration directly into the environment specific config-maps works initially, it puts a burden on the maintenance of that configuration as they are doomed to diverge over time. For such a situation the template approach is perfect. However, the setup with *Configuration Template* is more complicated and has more moving parts which can go south. Only use it if your configuration data is really large (like for Java EE application servers).

More information

- Popular template engines : Tiller, gomplate
- Example using gomplate and init-containers to prepare Wildfly configuration files from ConfigMap data.
- Init containers for doing one-shot initialisation before starting up an application

19. Immutable Configuration

Putting configuration data into containers which are linked to the application during runtime makes the configuration as immutable as the application itself.

Problem

As we have seen in the Env-Var Configuration pattern, environment variables provide a simplistic way to configure container based applications. And although they are easy to use and universally supported, as soon as the number of environments variables exceeds a certain threshold, managing them becomes unwieldily. This complexity can be handled with a Configuration Template, by using Configuration Resources or by looking up on a Configuration Service. However, all of these patterns do not enforce *immutability* of the configuration data. Immutability here means that we can't change the configuration after the application has started so that we always have a well-defined state for our configuration data. In additional immutable configuration can be put under version control so that an audit for configuration changes is easily possible.

Solution

The idea is to put all environment specific configuration data into a single, passive *data image* which we can distribute as a regular Docker image. During runtime, the application and the data image are linked together so that the application can extract the configuration from the data image. With this approach, it is easy to craft different configuration data images for various environments. These images then combine all configuration information for specific environments and can be versioned like any other Docker image.

Creating such a data image is trivial as it is a simple Docker image which contains only data. The challenge is the link step during startup. There are various approaches which depend on the platform support.

Docker Volumes

Before looking at Kubernetes let's go one step back and consider the vanilla Docker case. In Docker, it is possible for a container to expose a so-called *volume* with data from the container. With a `VOLUME` directive in a Dockerfile, you can specify a directory which can be shared later. During startup, the content of this directory within the container is copied over to this shared directory. This volume linking is a nice way for sharing configuration information from a dedicated configuration container with the application container.

Let's have a look at an example. For the development environment, we create a Docker image which holds developer configuration and creates a volume `/config`. We can create such an image with a Dockerfile `Dockerfile-config` :

```
FROM scratch

# Add the specified property
ADD app-dev.properties /config/app.properties

# Create volume and copy property into it
VOLUME /config
```

We now create the image itself and the Docker container with the Docker CLI:

```
docker build -t k8spatterns/config-dev-image:1.0.1 -f Dockerfile-con\
fig
docker create --name config-dev k8spatterns/config-dev-image:1.0.1 .
```

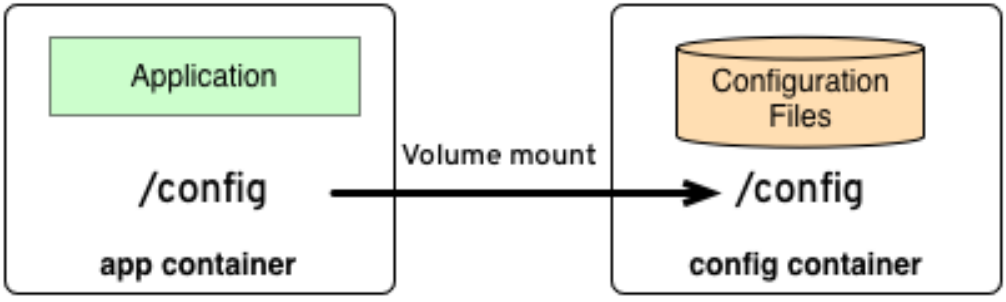
The final step is now to start the application container and connect it with this configuration container:

```
docker run --volumes-from config-dev k8spatterns/welcome-servlet:1.0
```

The application image expects its configuration files within a diretory `/config`, the volume exposed by the configuration container. When you now move this

application from the development environment to the production environment all you have to do is to change the startup command. There is no need to adapt the application image itself. Instead you simply volume link the application container with the production configuration container:

```
docker build -t k8spatterns/config-prod-image:1.0.1 -f Dockerfile-co\
nfig
docker create --name config-prod k8spatterns/config-prod-image:1.0.1\
 .
docker run --volumes-from config-prod k8spatterns/welcome-servlet:1.0
```



Immutable Configuration with Docker Volumes

Kubernetes Init-Containers

For Kubernetes volume sharing within a Pod is ideally suited for this kind of linking of configuration and application. However, if we want now to transfer this technique of Docker volume linking to the Kubernetes world, we discover soon, that there is currently no support for container volumes yet in Kubernetes. Considering the age of the discussion and the complexity to implement this feature versus its benefits it's likely that container volumes will not arrive ayntime soon though.

So containers can share (external) volumes, but they can not share yet directories located within the containers directly. To still use immutable configuration containers for application configuration init-containers can be used to initialize an empty shared volume during startup.

In the Docker example, we can base the configuration Docker image on *scratch* which is an empty Docker image without any operating system files. We didn't need anything more there because all that we want was the configuration data which is shared via Docker volumes. However, for Kubernetes we need some help from the base image to copy over the configuration data to a shared Pod volume. `busybox` is a good choice for the base image which is still small but allows us to use plain Unix `cp` for this task.

So how does the initialization of shared volumes with configuration works in detail?

Let's have a look at an example. First, we need to create a configuration image again with a Dockerfile like this:

```
FROM busybox

ADD dev.properties /config-src/demo.properties

# Using a shell here in order to resolve wildcards
ENTRYPOINT [ "sh", "-c", "cp /config-src/* $1", "--" ]
```

The only difference to the vanilla Docker case is that we have a different base image and that we added an `ENTRYPOINT` which copies the properties file to the directory given as argument when Docker image starts.

This image can now be referenced in an init-container within a `Deployment`'s template spec:

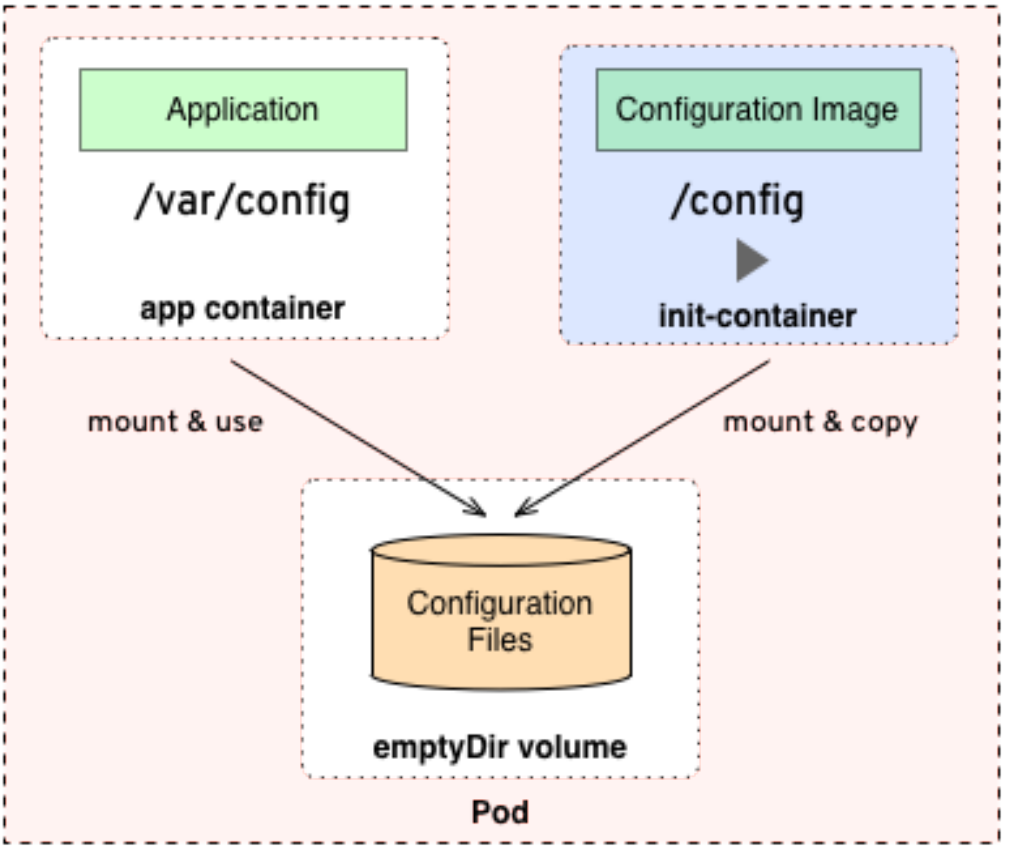
```
initContainers:
- image: k8spatterns/config-dev:1
  name: init
  args:
  - "/config"
  volumeMounts:
  - mountPath: "/config"
    name: config-directory
containers:
- image: k8spatterns/demo:1
```

```
11    name: demo
12    ports:
13    - containerPort: 8080
14      name: http
15      protocol: TCP
16    volumeMounts:
17    - mountPath: "/config"
18      name: config-directory
19  volumes:
20    - name: config-directory
21      emptyDir: {}
```

The `Deployment`'s pod template specification contains one volume and two containers: * The volume 'config-directory' is of type *emptyDir*, so it's created as an empty directory on the node hosting this pod. * The init-container which Kubernetes calls during startup. This init-container is built from our image that we just created above, and we set a single argument `/config` used by the image's entrypoint. This argument instructs the init-container to copy its content to the specified directory. The directory `/config` is mounted from the volume `config-directory`. * The application container mounts the volume `config-directory` to access the configuration which has been copied over by the init-container.

Now in order change the configuration from the development to the production environment all that we need to do is to exchange the image of the init-container. We can do this either manually or by an update with kubectl.

However this it is not ideal that we have to edit the resource descriptor for each environment. If you are on OpenShift, the enterprise edition of Kubernetes, then OpenShift Templates can help here.



Immutable Configuration with Init-Container

OpenShift Templates

Templates are regular resource descriptors which are parameterized. We can easily use the configuration image as a parameter:

```
apiVersion: v1
kind: Template
metadata:
  labels:
    project: k8spatterns
    pattern: ImmutableConfiguration
  name: demo
parameters:
  - name: CONFIG_IMAGE
    description: Image name of the configuration image to use
    value: k8spatterns/config-dev:1
objects:
- apiVersion: v1
  kind: DeploymentConfig
    // ....
    spec:
      template:
          metadata:
              // ....
            annotations:
                pod.beta.kubernetes.io/init-containers: |-
            [{
              "name": "init",
              "image": "${CONFIG_IMAGE}",
              "imagePullPolicy": "IfNotPresent",
              "args": [ "/config" ],
              "volumeMounts": [{
                "name": "config-directory",
                "mountPath": "/config"
              }]
            }]
            spec:
              containers:
                - image: k8spatterns/demo:1
                  // ...
```

```
                    volumeMounts:
                    - mountPath: "/config"
                      name: config-directory
            volumes:
            - name: config-directory
              emptyDir: {}
```

We show here only a fragment of the full descriptor. But we can easily recognize a parameter `CONFIG_IMAGE` which we reference in the init-container declaration. This time the init-container is declared as a particular pod annotation since OpenShift at the time of the writing does not support yet the 'initContainers:' syntax from newer Kubernetes versions yet.

If we create this template on an OpenShift cluster we can instantiate it by

```
oc new-app demo -p CONFIG_IMAGE=k8spatterns/config-prod:1
```

Detailed instructions, as well as the full deployment descriptors, can be found in our example repository.

Discussion

Using data containers for the Immutable Configuration pattern is admittedly a bit involved. However, this pattern has some unique advantages:

- Environment-specific configuration is sealed within in container. Therefore it can be versioned like any other Docker image.
- Configuration created this way can be distributed over a Docker registry. The configuration can be examined even without accessing the cluster.
- The configuration is immutable as the Docker image holding the configuration: A change in the configuration requires a version update and new Docker image.
- Configuration data images are useful when the configuration data is too complex to stuff into environment variables or `ConfigMaps` since it can hold arbitrary large configuration data.

Of course, there are also certain drawbacks for this pattern:

- It has a higher complexity because extra images needs to be built and distributed via registries
- Extra init-container processing is required in the Kubernetes case and hence we need to manage different `Deployment` objects.

All in all, it should be carefully evaluated whether such an involved approach is required. Maybe a simple `ConfigMap` as described in ConfigurationResource is completely sufficient, too, if mutability is not a thing for you.

More information

- Full working examples for this pattern including installation instructions.
- Stack Overflow answer how to map the Docker volume concept to Kubernetes.
- Long lasting GitHub issue about how to mimic the Docker behaviour with a dedicated volume type for data-containers.

V Advanced Patterns

The patterns in this category are implemented by various projects and frameworks running on top of Kubernetes. But these patterns represent generic enough logic that can be implemented for custom application use cases too.

20. Stateful Service

Problem

Solution

Discussion

More information

21. Custom Controller

Problem

Solution

Discussion

More information

22. Build Container

Problem

Solution

Discussion

More information