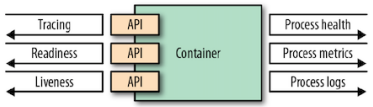
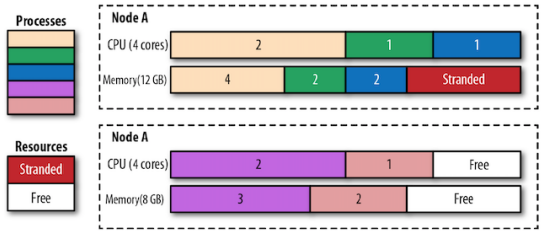
Top 10 Must-Know Design Patterns for Kubernetes Beginners

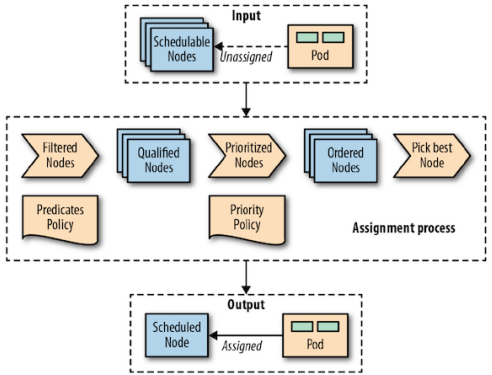
Foundational



Health Probe

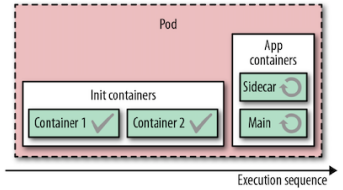


Predictable Demands

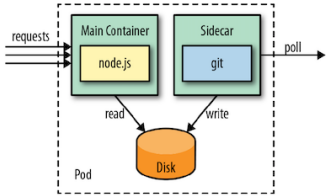


Automated Placement

Structural

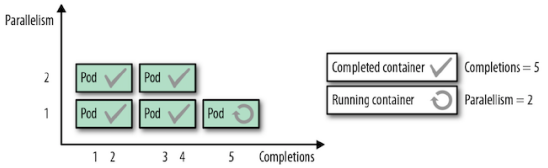


Init Container

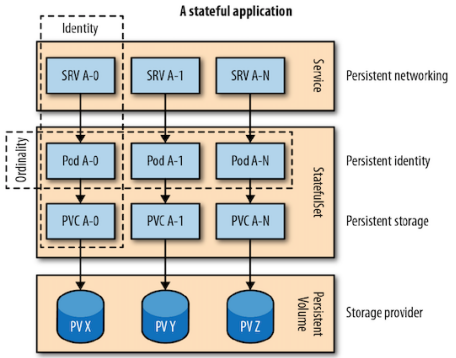


Sidecar

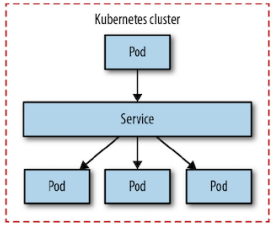
Behavioural



Batch Job

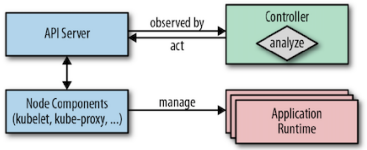


Stateful Service

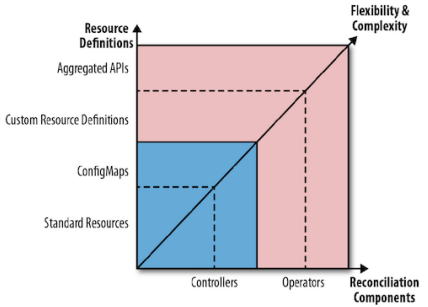


Service Discovery

Higher-level



Controller



Operator

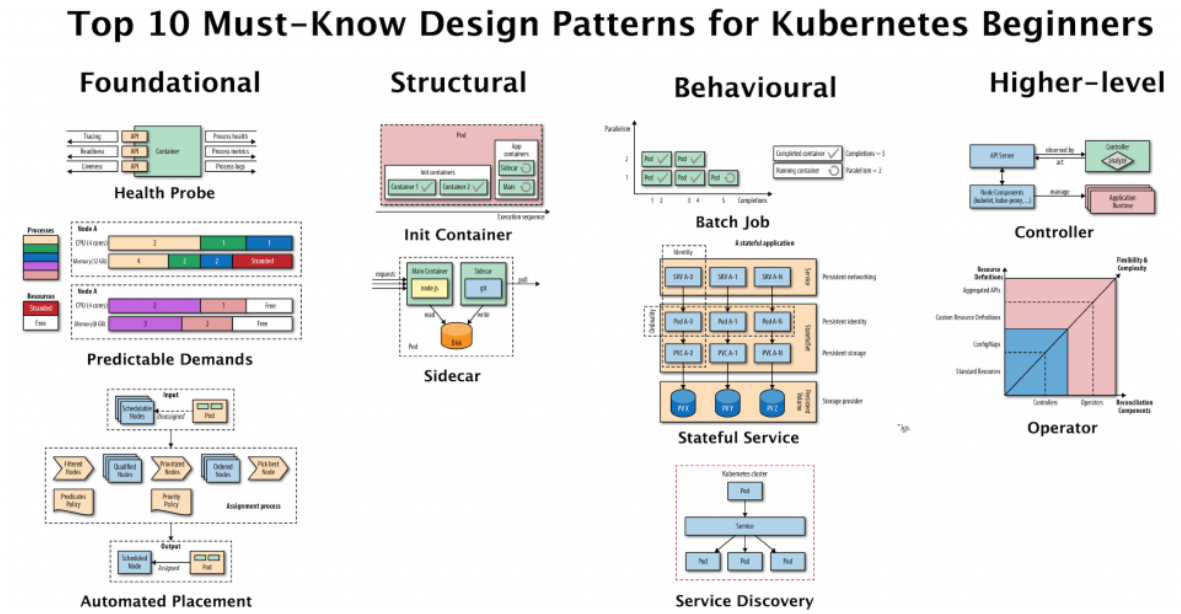
Article

Top 10 must-know Kubernetes design patterns

May 11, 2020 Containers, DevOps, Kubernetes, Operators

Bilgin Ibryam

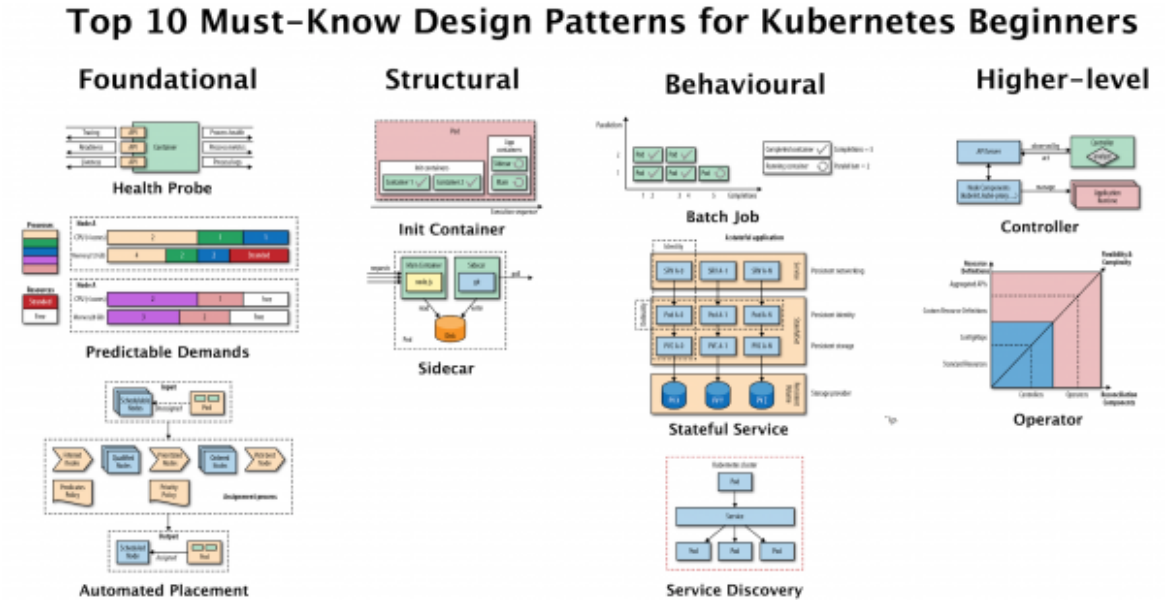
Red Hat Product Manager



Top 10 Kubernetes Design Patterns

Here are the must-know top 10 design patterns for beginners synthesized from the Kubernetes Patterns book. Getting familiar with these patterns will help you understand foundational Kubernetes concepts, which in turn will help you in discussions and when designing Kubernetes-based applications.

There are many important concepts in Kubernetes, but these are the most important ones to start with:



Top 10 Kubernetes Design Patterns

To help you understand, the patterns are organized into a few categories below, inspired by the Gang of Four's design patterns.

Foundational patterns

These patterns represent the principles and best practices that containerized applications must comply with in order to become good cloud-native citizens. Regardless of the application's nature, you should aim to follow these guidelines. Adhering to these principles will help ensure that your applications are suitable for automation on Kubernetes.

Health Probe pattern

Health Probe dictates that every container should implement specific APIs to help the platform observe and manage the application in the healthiest way possible. To be fully automatable, a cloud-native application must be highly observable by allowing its state to be inferred so that Kubernetes can detect whether the application is up and ready to serve requests. These observations influence the life-cycle management of Pods and the way traffic is routed to the application.

Predictable Demands pattern

Predictable Demands explains why every container should declare its resource profile and stay confined to the indicated resource requirements. The foundation of successful application deployment, management, and coexistence on a shared cloud environment is dependent on identifying and declaring the application's resource requirements and runtime dependencies. This pattern describes how you should declare application requirements, whether they are hard runtime

dependencies or resource requirements. Declaring your requirements is essential for Kubernetes to find the right place for your application within the cluster.

Automated Placement patterns

Automated Placement explains how to influence workload distribution in a multi-node cluster. Placement is the core function of the Kubernetes scheduler for assigning new Pods to nodes satisfying container resource requests and honoring scheduling policies. This pattern describes the principles of Kubernetes' scheduling algorithm and the way to influence the placement decisions from the outside.

Structural patterns

Having good cloud-native containers is the first step, but not enough. Reusing containers and combining them into Pods to achieve the desired outcome is the next step. The patterns in this category are focused on structuring and organizing containers in a Pod to satisfy different use cases. The forces that affect containers in Pods result in these patterns.

Init Container pattern

Init Container introduces a separate life cycle for initialization-related tasks and the main application containers. Init Containers enable separation of concerns by providing a separate life cycle for initialization-related tasks distinct from the main application containers. This pattern introduces a fundamental Kubernetes concept that is used in many other patterns when initialization logic is required.

Sidecar patterns

Sidecar describes how to extend and enhance the functionality of a pre-existing container without changing it. This pattern is one of the fundamental container patterns that allows single-purpose containers to cooperate closely together.

Behavioral patterns

These patterns describe the life-cycle guarantees of the Pods ensured by the managing platform. Depending on the type of workload, a Pod might run until completion as a batch job or be scheduled to run periodically. It might run as a daemon service or singleton. Picking the right life-cycle management primitive will help you run a Pod with the desired guarantees.

Batch Job patterns

Batch Job describes how to run an isolated, atomic unit of work until completion. This pattern is suited for managing isolated atomic units of work in a distributed environment.

Stateful Service patterns

Stateful Service describes how to create and manage distributed stateful applications with Kubernetes. Such applications require features such as persistent identity, networking, storage, and ordinality. The StatefulSet primitive provides these building blocks with strong guarantees ideal for the management of stateful applications.

Service Discovery pattern

Service Discovery explains how clients can access and discover the instances that are providing application services. For this purpose, Kubernetes provides multiple mechanisms, depending on whether the service consumers and producers are located on or off the cluster.

Higher-level patterns

The patterns in this category are more complex and represent higher-level application management patterns. Some of the patterns here (such as Controller) are timeless, and Kubernetes itself is built on top of them.

Controller pattern

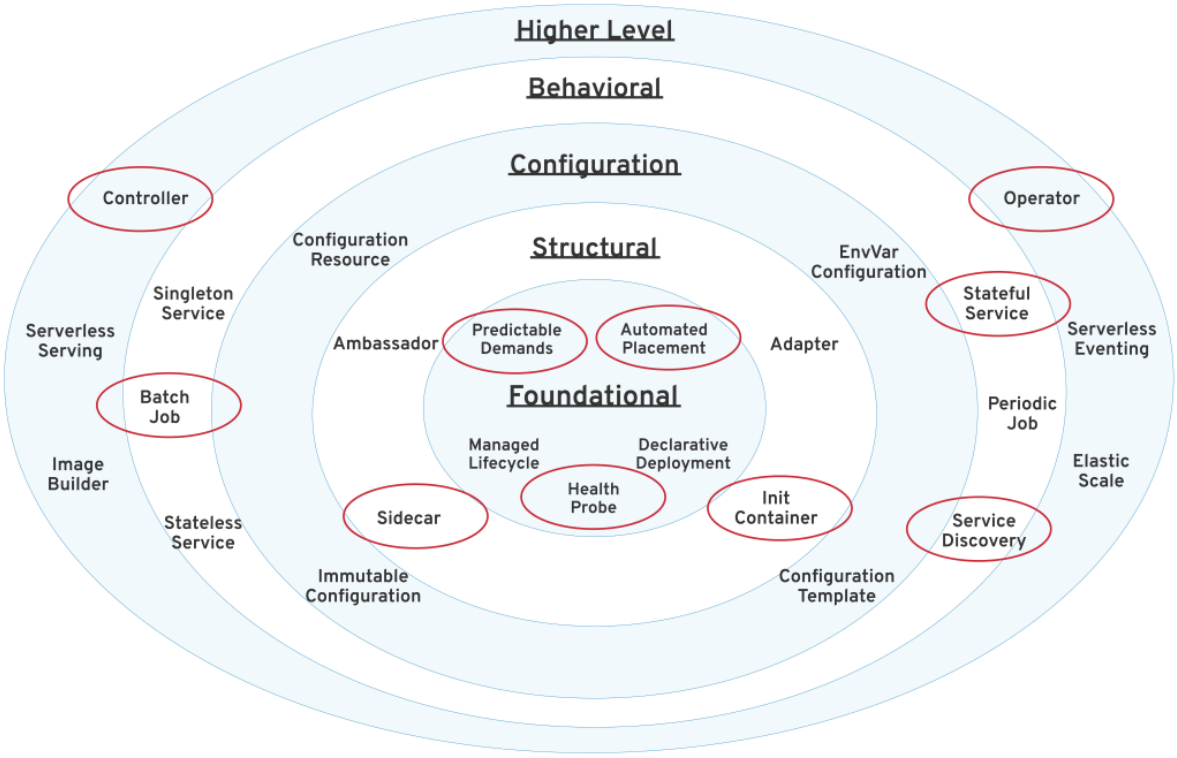
Controller is a pattern that actively monitors and maintains a set of Kubernetes resources in a desired state. The heart of Kubernetes itself consists of a fleet of controllers that regularly watch and reconcile the current state of applications with the declared target state. This pattern describes how to leverage this core concept for extending the platform for our own applications.

Operator pattern

An *Operator* is a Controller that uses a CustomResourceDefinitions to encapsulate operational knowledge for a specific application in an algorithmic and automated form. The Operator pattern allows us to extend the Controller pattern for more flexibility and greater expressiveness. There are an increasing number of Operators for Kubernetes, and this pattern is turning into the major form of operating complex distributed systems.

Summary

Today, Kubernetes is the most popular container orchestration platform. It is jointly developed and supported by all major software companies and offered as a service by all of the major cloud providers. Kubernetes supports both Linux and Windows systems, plus all major programming languages. This platform can also orchestrate and automate stateless and stateful applications, batch jobs, periodic tasks, and serverless workloads. The patterns described here are the most commonly used ones from a broader set of patterns that come with Kubernetes as shown below.



Kubernetes Patters organized in different categories

Kubernetes is the new application portability layer and the common denominator among everybody on the cloud. If you are a software developer or architect, the odds are that Kubernetes will become part of your life in one form or another. Learning about the Kubernetes patterns described here will change the way you think about this platform. I believe that Kubernetes and the concepts originating from it will become as fundamental as object-oriented programming concepts.

The patterns here are an attempt to create the Gang of Four design patterns, but for container orchestration. Reading this article must not be the end, but the beginning of your Kubernetes journey. Happy `kubectl` -ing!

Last updated: June 26, 2020

Recent Articles