4.1. Design

4.3. Learning Objectives

By the end of this chapter, you should be able to:

- Define an application's resource requirements.
- Understand multi-container pod design patterns: Ambassador, Adapter, Sidecar.
- Discuss application design concepts.

4.4. Traditional Applications - Considerations

One of the larger hurdles towards implementing Kubernetes in a production environment is the suitability of application design. Optimal Kubernetes deployment design changes are more than just the simple containerization of an application. Traditional applications were built and deployed with the expectation of long-term processes and strong interdependence.

For example, an Apache web server allows for incredible customization. Often, the server would be configured and tweaked without interruption. As demand grows, the application may be migrated to larger and larger servers. The build and maintenance of the application assumes the instance would run without reset and have persistent and tightly coupled connections to other resources, such as networks and storage.

In early usage of containers, applications were containerized without redevelopment. This lead to issues after resource failure, or upon upgrade, or configuration. The cost and hassle of redesign and re-implementation should be taken into account.

4.5. Decoupled Resources

The use of decoupled resources is integral to Kubernetes. Instead of an application using a dedicated port and socket, for the life of the instance, the goal is for each component to be decoupled from other resources. The expectation and software development toward separation allows for each component to be removed, replaced, or rebuilt.

Instead of hard-coding a resource in an application, an intermediary, such as a Service, enables connection and reconnection to other resources, providing flexibility. A single machine is no longer required to meet the application and user needs; any number of systems could be brought together to meet the needs when, and, as long as, necessary.

As Kubernetes grows, even more resources are being divided out, which allows for an easier deployment of resources. Also, Kubernetes developers can optimize a particular function with fewer considerations of others objects.

4.6. Transience

Equally important is the expectation of transience. Each object should be developed with the expectation that other components will die and be rebuilt. With any and all resources planned for transient relationships to others, we can update versions or scale usage in an easy manner.

An upgrade is perhaps not quite the correct term, as the existing application does not survive. Instead, a controller terminates the container and deploys a new one to replace it, using a different version of the application or setting. Typically, traditional applications were not written this way, opting toward long-term relationships for efficiency and ease of use.

4.7. Flexible Framework

Like a school of fish, or pod of whales, multiple independent resources working together, but decoupled from each other and without expectation of individual permanent relationship, gain flexibility, higher availability and easy scalability. Instead of a monolithic Apache server, we can deploy a flexible number of **nginx** servers, each handling a small part of the workload. The goal is the same, but the framework of the solution is distinct.

A decoupled, flexible and transient application and framework is not the most efficient. In order for the Kubernetes orchestration to work, we need a series of agents, otherwise known as controllers or watch-loops, to constantly monitor the current cluster state and make changes until that state matches the declared configuration.

The commoditization of hardware has enabled the use of many smaller servers to handle a larger workload, instead of single, huge systems.

4.8. Managing Resource Usage

As with any application, an understanding of resource usage can be helpful for a successful deployment. Kubernetes allows us to easily scale clusters, larger or smaller, to meet demand. An understanding of how the Kubernetes clusters view the resources is an important consideration. The **kube-scheduler**, or a custom scheduler, uses the ***PodSpec*** to determine the best node for deployment.

In addition to administrative tasks to grow or shrink the cluster or the number of Pods, there are *autoscalers* which can add or remove nodes or pods, with plans for one which uses cgroup settings to limit CPU and memory usage by individual containers.

By default, Pods use as much CPU and memory as the workload requires, behaving and coexisting with other Linux processes. Through the use of resource *requests*, the scheduler will only schedule a Pod to a node if resources exist to meet all requests on that node. The scheduler takes these and several other factors into account when selecting a node for use.

Monitoring the resource usage cluster-wide is not an included feature of Kubernetes. Other projects, like Prometheus, are used instead. In order to view resource consumption issues locally, use the **`kubectl describe pod`** command. You may only know of issues after the pod has been terminated.

4.15. Ambassador

An ambassador allows for access to the outside world without having to implement a service or another entry in an ingress controller:

- Proxy local connection
- Reverse proxy
- Limits HTTP requests
- Re-route from the main container to the outside world.

Ambassador is an "*open source, Kubernetes-native API gateway for microservices built on Enjoy*".

4.16. Points to Ponder

A few things to carefully consider:

- Is my application as decoupled as it could possibly be? Is there anything that I could take out, or make its own container?
- Is each container transient, does it expect others to be transient?
- Can I scale any particular component to meet workload demand?
- Have I used the most open standard stable enough to meet my needs?

4.9. CPU

CPU requests are made in CPU units, each unit being a millicore, using mille - the Latin word for thousand. Some documentation uses *millicore*, others use *millicpu*, but both have the same meaning. Thus, a request for .7 of a CPU would be 700 milicores. Should a container use more resources than allowed, it won't be killed. The exact amount of overuse is not definite.

- **`spec.containers[].resources.limits.cpu`**
- **`spec.containers[].resources.requests.cpu`**

The value of CPUs is not relative. It does not matter how many exist, or if other Pods have requirements. One CPU, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading.

4.10. Memory (RAM)

With Docker engine, the **limits.memory** value is converted to an integer value and becomes the value to the **docker run --memory <value> <image>** command. The handling of a container which exceeds its memory limit is not definite. It may be restarted, or, if it asks for more than the memory request setting, the entire Pod may be evicted from the node.

- **spec.containers[].resources.limits.memory**
- **spec.containers[].resources.requests.memory**

4.11. Ephemeral Storage (Alpha Feature)

Container files, logs, and *EmptyDir* storage, as well as Kubernetes cluster data, reside on the root filesystem of the host node. As storage is a limited resource, you may need to manage it as other resources. The scheduler will only choose a node with enough space to meet the sum of all the container requests. Should a particular container, or the sum of the containers in a Pod, use more then the limit, the Pod will be evicted.

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

4.12. Multi-Container Pods

The idea of multiple containers in a Pod goes against the architectural idea of decoupling as much as possible. One could run an entire operating system inside a container, but would lose much of the granular scalability Kubernetes is capable of. But there are certain needs in which a second or third co-located container makes sense. By adding a second container, each container can still be optimized and developed independently, and both can scale and be repurposed to best meet the needs of the workload.

4.13. Sidecar Container

The idea for a *sidecar container* is to add some functionality not present in the main container. Rather than bloating code, which may not be necessary in other deployments, adding a container to handle a function such as logging solves the issue, while remaining decoupled and scalable. Prometheus monitoring and Fluentd logging leverage sidecar containers to collect data.

4.14. Adapter Container

The basic purpose of an *adapter container* is to modify data, either on ingress or egress, to match some other need. Perhaps, an existing enterprise-wide monitoring tools has particular data format needs. An adapter would be an efficient way to standardize the output of the main container to be ingested by the monitoring tool, without having to modify the monitor or the containerized application. An adapter container transforms multiple applications to singular view.