Bezpieczeństwo infrastruktury krytycznej i systemów sterowania przemysłowego IoT - Laboratorium 4 – Kubernetes, Helm

Forma: praca w samodzielna / małe zespoły (2 - 4 osoby)

Wstęp teoretyczny

Pod

https://kubernetes.io/pl/docs/concepts/workloads/pods/

Pod jest najmniejszą jednostką obliczeniową, którą można utworzyć i zarządzać nią w Kubernetesie.

Pod (w języku angielskim: jak w odniesieniu do grupy wielorybów lub strąka grochu) to grupa jednego lub więcej kontenerów, z współdzielonymi zasobami pamięci i sieci, oraz specyfikacją dotyczącą sposobu uruchamiania kontenerów. Wszystkie komponenty Poda są uruchamiane razem, współdzielą ten sam kontekst i są planowane do uruchomienia na tym samym węźle. Pod modeluje specyficznego dla aplikacji "logicznego hosta": zawiera jeden lub więcej kontenerów aplikacji, które są stosunkowo ściśle ze sobą powiązane. W kontekstach niechmurowych, aplikacje wykonane na tej samej maszynie fizycznej lub wirtualnej są analogiczne do aplikacji chmurowych wykonanych na tym samym logicznym hoście.

Oprócz kontenerów aplikacyjnych, Pod może zawierać kontenery inicjalizujące uruchamiane podczas startu Pod. Możesz również wstrzyknąć kontenery efemeryczne do debugowania działającego Poda.

Czym jest Pod?

Wspólny kontekst Poda to zestaw przestrzeni nazw Linux, cgroups i potencjalnie innych aspektów izolacji - te same elementy, które izolują kontener (ang. container). W obrębie kontekstu Poda, poszczególne aplikacje mogą mieć dodatkowo zastosowane dalsze sub-izolacje.

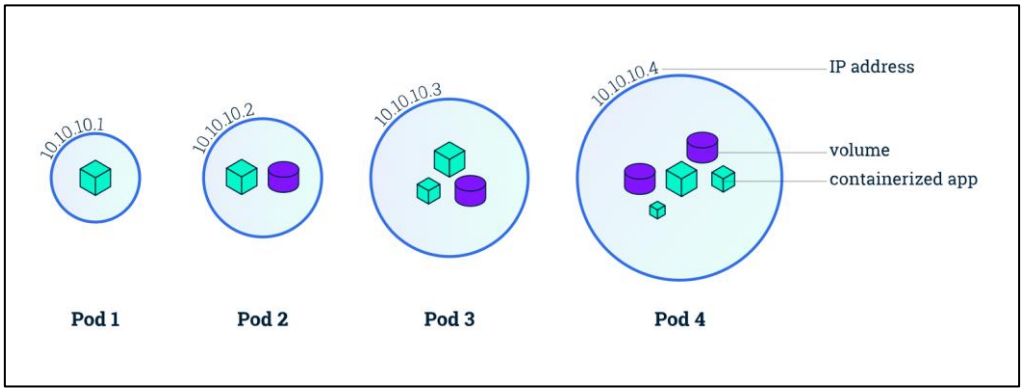
Pod jest podobny do zestawu kontenerów z współdzielonymi przestrzeniami nazw i współdzielonymi woluminami systemu plików.

Pody w klastrze Kubernetesa są używane na dwa główne sposoby:

- **Pody, które uruchamiają pojedynczy kontener**. Model "jeden-kontener-na-Poda" jest najczęstszym przypadkiem użycia; w tym przypadku możesz myśleć o Podzie jako o obudowie wokół pojedynczego kontenera; Kubernetes zarządza Podami, zamiast zarządzać kontenerami bezpośrednio.
- **Pody, które uruchamiają wiele kontenerów, które muszą współdziałać**. Pod może zawierać aplikację składającą się z wielu współlokalizowanych kontenerów, które są ściśle powiązane i muszą współdzielić zasoby. Te współlokalizowane kontenery tworzą jedną spójną jednostkę.

Grupowanie wielu współlokalizowanych i współzarządzanych kontenerów w jednym Podzie jest stosunkowo zaawansowanym przypadkiem użycia. Ten wzorzec powinieneś używać tylko w określonych przypadkach, gdy twoje kontenery są ściśle powiązane.

Nie musisz uruchamiać wielu kontenerów, aby zapewnić replikację (dla odporności lub pojemności), jeśli potrzebujesz wielu replik, zobacz zarządzanie workloadami.



Node

https://kubernetes.io/pl/docs/tutorials/kubernetes-basics/explore/explore-intro/

W kontekście Kubernetes, node to pojedyncza maszyna fizyczna lub wirtualna, która działa jako część klastra Kubernetes. Klastry te składają się z jednego lub wielu węzłów, a każdy z nich pełni rolę w hostowaniu, uruchamianiu i zarządzaniu kontenerami. Kontenery to zizolowane jednostki oprogramowania, które zawierają kod, zależności i wszystko, co jest potrzebne do poprawnego działania aplikacji.

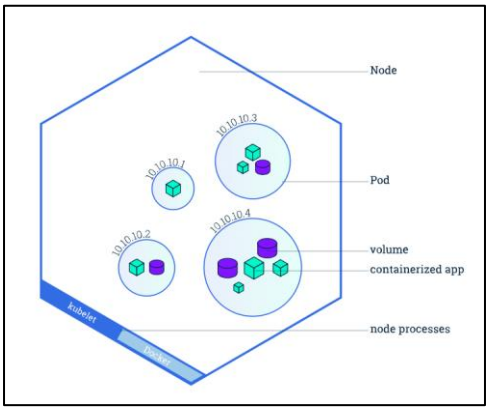
Node składa się z kilku głównych komponentów:

1. Kubelet: Jest to agent działający na każdym węźle w klastrze, komunikujący się z API serwerem Kubernetes. Jego głównym zadaniem jest utrzymanie kontenerów na danym węźle oraz zarządzanie cyklem życia kontenerów.
2. Kube-proxy: Odpowiada za zarządzanie ruchem sieciowym w klastrze. Tworzy ona przekierowania (proxy) do usług działających w klastrze, umożliwiając im komunikację zewnętrzną.
3. Container runtime: Jest to oprogramowanie, które jest odpowiedzialne za uruchamianie kontenerów. Popularne kontenery runtime to Docker czy containerd.

Rola Nodes w Kubernetes

Nodes w Kubernetes pełnią kluczową rolę w procesie wdrażania i zarządzania aplikacjami. Oto kilka kluczowych aspektów ich roli:

1. **Hosting Kontenerów:** Nodes są miejscem, gdzie faktycznie uruchamiane są kontenery. Mogą to być zarówno aplikacje użytkowe, jak i komponenty systemowe, takie jak bazy danych czy serwery webowe.
2. **Zarządzanie Zasobami:** Nodes zarządzają dostępnymi zasobami, takimi jak CPU, pamięć RAM czy dysk. Kubernetes używa tych informacji do efektywnego planowania i alokacji zasobów dla aplikacji.
3. **Komunikacja i Sieć:** Kube-proxy na każdym węźle zarządza ruchem sieciowym. Nodes komunikują się ze sobą i umożliwiają aplikacjom w klastrze kontakt z zewnętrznym światem.
4. **Skalowanie i Przetrzymywanie Aplikacji:** Nodes są podstawową jednostką, która może być skalowana w klastrze. Wraz z rozwojem obciążenia, nowe węzły mogą być dodawane, a istniejące mogą być usuwane.
5. **Monitorowanie i Diagnostyka:** Kubelet zbiera dane o stanie kontenerów i przesyła je do API serwera. Dzięki temu, narzędzia monitorujące i diagnostyczne mogą analizować i reagować na zmiany w klastrze.



Namespace

https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

W Kubernetes namespaces (przestrzenie nazw) zapewniają mechanizm izolowania grup zasobów w ramach jednego klastra. Nazwy zasobów muszą być unikalne wewnątrz jednej przestrzeni nazw, ale nie muszą być unikalne pomiędzy różnymi przestrzeniami nazw. Zakres oparty na przestrzeniach nazw dotyczy tylko obiektów, które są namespaced (np. *Deployments*, *Services* itp.), a nie obiektów o zasięgu całego klastra (np. *StorageClass*, *Nodes*, *PersistentVolumes* itp.).

Kiedy używać wielu przestrzeni nazw

Przestrzenie nazw są przeznaczone do użycia w środowiskach, w których pracuje wielu użytkowników rozproszonych pomiędzy różnymi zespołami lub projektami. W klastrach z niewielką liczbą użytkowników (kilku lub kilkunastu) nie ma potrzeby tworzenia ani używania przestrzeni nazw. Zacznij korzystać z przestrzeni nazw dopiero wtedy, gdy potrzebujesz funkcji, które one zapewniają.

Zakres i unikalność nazw

Przestrzenie nazw definiują zakres dla nazw zasobów. Nazwy zasobów muszą być unikalne w obrębie jednej przestrzeni nazw, ale nie muszą być unikalne pomiędzy różnymi przestrzeniami nazw. Przestrzenie nazw nie mogą być zagnieżdżane, a każdy zasób Kubernetes może należeć tylko do jednej przestrzeni nazw. Przestrzenie nazw stanowią sposób podziału zasobów klastra pomiędzy wielu użytkowników (np. za pomocą resource quota - limitów zasobów). Nie jest konieczne używanie wielu przestrzeni nazw do oddzielania nieznacznie różnych zasobów, np. różnych wersji tego samego oprogramowania - w takim przypadku należy użyć etykiet (labels) w obrębie tej samej przestrzeni nazw.

Początkowe przestrzenie nazw

Kubernetes uruchamia się z czterema początkowymi przestrzeniami nazw:

- **default**
  Kubernetes zawiera tę przestrzeń nazw, aby można było zacząć korzystać z nowego klastra bez konieczności wcześniejszego jej tworzenia.

- **kube-node-lease**
  Ta przestrzeń nazw przechowuje obiekty typu *Lease* powiązane z każdym węzłem.
  Node leases umożliwiają *kubeletowi* wysyłanie sygnałów heartbeat, dzięki czemu *control plane* może wykrywać awarie węzłów.

- **kube-public**
  Ta przestrzeń nazw jest czytelna dla wszystkich klientów (w tym nieautoryzowanych).
  Jest głównie zarezerwowana do użytku klastrowego — na wypadek, gdyby pewne zasoby miały być publicznie widoczne i dostępne w całym klastrze.
  Publiczny charakter tej przestrzeni nazw jest tylko konwencją, a nie wymogiem.

- **kube-system**
  Przestrzeń nazw przeznaczona dla obiektów tworzonych przez system Kubernetes.

Klastry

https://kubernetes.io/pl/docs/concepts/overview/components/

https://kubernetes.io/pl/docs/concepts/architecture/

Ogólną, a jednocześnie bardzo trafną odpowiedzią na to pytanie jest następująca definicja:

Klaster to rodzaj równoległego lub rozproszonego systemu, który:

- składa się ze zbioru połączonych pełnych komputerów oraz
- używany jest jako pojedynczy system (komputer).

Istotne jest tu użycie terminu pełny komputer - chodzi o komputer składający się z jednej lub więcej jednostek obliczeniowych, odpowiedniej ilości pamięci, podsystemu wejścia-wyjścia oraz systemu operacyjnego. Istnieje wiele systemów mogących przypominać klaster, które w myśl powyższej definicji nim nie są. Na przykład niektóre systemy typu NUMA (ang. *Non-Uniform Memory Access*) składają się z niezależnych komputerów, ale pracujących pod kontrolą jednej instancji systemu operacyjnego.

Zazwyczaj w innych definicjach można się spotkać z określeniem klastra jako systemu sprawiającego wrażenie pojedynczego systemu (ang. *single system image*). Autor pierwszej przytoczonej definicji celowo posłużył się określeniem ,,używany''. Nie na wszystkich poziomach funkcjonalności system rzeczywiście musi sprawiać wrażenie pojedynczego systemu. Przykładowo z punktu widzenia końcowego użytkownika system może wydawać się pojedynczym systemem: aplikacja użytkownika wysyła żądania do klastra i z klastra otrzymuje odpowiedzi, nie interesując się, który z węzłów obsłużył żądanie. Z punktu widzenia administratora systemu wygląda to inaczej - może on mieć (i z reguły ma) do czynienia z niezależnymi systemami.

W powyższej definicji klaster określa się jako rodzaj systemu rozproszonego bądź równoległego, nie precyzuje jednak na czym te różnice polegają. Różnice w stosunku do systemów równoległych typu SMP (ang. *Symmetric Multi-Processor*) są dość oczywiste. Po pierwsze jest to skalowalność, bo łatwiej rozbudować klaster niż system typu SMP. Dodając nowy węzeł dodaje się pełny komputer, a więc podsystem wejścia-wyjścia, pamięć itp. W przypadku SMP trudno dodawać kolejne procesory w nieskończoność bez zmian w innych częściach systemu. Po drugie jest to dostępność - awaria jednego procesora w SMP lub podsystemu oznacza zazwyczaj unieruchomienie całego systemu. W klastrze awaria węzła nie powinna przerywać jego pracy. Po trzecie - łatwość zarządzania. Na poziomie systemu operacyjnego SMP jest pojedynczym komputerem, zarządzanie nim jest prostsze niż systemem rozproszonym, jakim jest system klastrowy. Inną może mniej oczywistą różnicą jest kwestia licencji na oprogramowanie. W przypadku klastra z reguły oznacza to konieczność wykupienia tylu licencji na oprogramowanie (choćby na system operacyjny), z ilu węzłów składa się klaster.

Różnice w stosunku do typowych systemów rozproszonych są mniej oczywiste. Pierwszą z nich jest anonimowość węzłów. W ogólnie rozumianym systemie rozproszonym, każdy jego element jest rozróżnialny. W klastrach natomiast dąży się do rozwiązań polegających na jak największej anonimowości węzłów. Nie znaczy to, że każdy z węzłów musi być identyczny i pełnić dokładnie takie same funkcje, chodzi bowiem o wspomniane używanie klastra jako pojedynczego systemu.

Drugą różnicą są wzajemne relacje pomiędzy węzłami systemu z uwzględnieniem dostępu do danych. W systemach rozproszonych najczęściej stosuje się paradygmat klient-serwer. Innymi słowy, istnieje zróżnicowanie funkcjonalne i hierarchiczna zależność pomiędzy węzłami, nawet jeśli fizycznie węzły połączone są jak równy-z-równym (ang. *peer-to-peer*). Przykładowo klienci rozproszonego systemu plików nie wymieniają danych bezpośrednio, komunikują się za pośrednictwem serwera (bądź serwerów). W systemach klastrowych hierarchiczna struktura z uwagi na dostęp do danych nie występuje, nawet jeśli istnieją różnice funkcjonalne pomiędzy węzłami. Każdy z węzłów klastra ma dostęp do wszystkich (lub prawie wszystkich) danych w systemie. Niespełnienie tego warunku może przeczyć takim cechom klastrów, jak dystrybucja zadań czy skalowalność.

Warto zwrócić uwagę, że równouprawniony dostęp do danych w klastrach ma miejsce zarówno w przypadku klastrów dzielących dane (ang. *shared data*), których wszystkie węzły fizycznie są podłączone do pamięci stałej, jak i takich, które nie dzielą pamięci stałej (ang. *shared-nothing*). W drugim przypadku węzeł, chcący uzyskać dostęp do danych, które nie są osiągalne lokalnie, przekazuje żądanie do węzła, które je posiada.

Mikroserwisy

https://microservices.io/

Mikroserwisy (lub mikrousługi) to niewielkie usługi, których przygotowanie zajmuje maksymalnie kilka tygodni. Obejmują tylko jeden obszar biznesu. Ideą mikroserwisu jest podział monolitycznej aplikacji na zbiór mniejszych usług, które pozostają ze sobą w relacji. Każdy mikroserwis jest *de facto* małą aplikacją. Ma własną architekturę, złożoną z logiki biznesowej i różnych adapterów.

W porównaniu z monolitem mikrousługi mają wiele zalet:

- pozwalają lepiej kontrolować poziom długu technicznego
- redukują ryzyko zatrzymania pracy całego systemu z powodu drobnego błędu w jednym obszarze
- umożliwiają szybsze i częstsze wdrożenia nowych wersji
- podnoszą stabilność systemu

Architektura mikroserwisowa wydaje się idealna zwłaszcza, gdy trzeba często testować pomysły na nowe funkcjonalności i sprawdzać, jak zareagują na nie użytkownicy.

Zalety architektury mikroserwisowej

- adresuje problem złożoności, bo rozkłada aplikację na wiele łatwych w zarządzaniu elementów, znacznie szybszych w opracowaniu i łatwiejszych w zrozumieniu i utrzymaniu
- pozwala niezależnie rozwijać każdą usługę przez zespół, który dzięki temu koncentruje się tylko na funkcjonalności tej usługi
- umożliwia swobodny wybór technologii
- oznacza niezależne i szybsze wdrożenie każdej usługi
- pozwala niezależnie skalować każdą usługę

Wady

Architektura mikrousługowa jest bardziej skomplikowana, bo jest architekturą systemów rozproszonych. Komplikuje projekt, między innymi powoduje, że musimy wybrać i wdrożyć mechanizm komunikacji między procesami oparty na komunikatach lub RPC. Musimy też oprogramować wyjątki i awarie oraz wziąć pod uwagę pozostałe aspekty obliczeń rozproszonych. Ponadto:

- architektura partycjonowanej bazy danych. Transakcje biznesowe aktualizują wiele jednostek, w aplikacji opartej na mikroserwisach aktualizują wiele baz danych należących do różnych usług. Korzystanie z transakcji rozproszonych zwykle nie wchodzi w rachubę i ostatecznie musimy zapewnić spójność programowo, co jest oczywiście trudniejsze dla programistów
- testowanie aplikacji opartych na mikrousługach jest znacznie bardziej złożone niż monolitycznej aplikacji webowej. Musimy uruchomić tę usługę i wszystkie usługi powiązane
- trudniej jest wprowadzić zmiany obejmujące wiele usług. W monolitycznej aplikacji możemy po prostu zmienić odpowiednie moduły, zintegrować zmiany i wdrożyć je za jednym razem. W architekturze mikrousługowej dokładnie planujemy i koordynujemy wdrażanie zmian w każdej z usług
- wdrażanie aplikacji jest też bardziej złożone. Monolityczna aplikacja jest po prostu wdrażana z wykorzystaniem zestawu maszyn wirtualnych. Natomiast aplikacja oparta na mikrousługach zazwyczaj składa się z wielu usług, a każda ma wiele instancji środowiska wykonawczego. Z kolei każda instancja musi być skonfigurowana, wdrożona, skalowana i monitorowana. Musimy także zapewnić mechanizm wykrywania usług. Ten poziom złożoności wymaga już dużej automatyzacji

Service

https://kubernetes.io/docs/concepts/services-networking/service/

Service udostępnia aplikację działającą w Twoim klastrze pod jednym zewnętrznym punktem dostępowym, nawet jeśli obciążenie jest rozproszone między wiele backendów. W Kubernetes Service to mechanizm służący do udostępniania aplikacji sieciowej działającej jako jeden lub więcej Podów w klastrze.

Cel istnienia Service w Kubernetes

Kluczowym celem Service w Kubernetes jest to, że nie musisz modyfikować istniejącej aplikacji, aby korzystała z nowego, nieznanego mechanizmu odkrywania usług (*service discovery*). Możesz uruchamiać kod w Podach, niezależnie od tego, czy jest to kod zaprojektowany dla środowiska *cloud-native*, czy starsza aplikacja, którą skonteneryzowałeś. Używasz Service, aby udostępnić zestaw Podów w sieci — tak, by klienci mogli się z nimi komunikować.

Podstawowy problem: dynamiczność Podów

Jeśli używasz **Deployment** do uruchamiania aplikacji, Deployment może dynamicznie tworzyć i usuwać Pody.
W danej chwili nie wiesz dokładnie:

- ile Podów działa i jest zdrowych,
- ani jakie mają nazwy.

Kubernetes tworzy i usuwa Pody, aby utrzymać pożądany stan klastra. Pody są **efemeryczne** - nie należy oczekiwać, że pojedynczy Pod będzie trwały czy niezawodny. Każdy Pod otrzymuje własny adres IP (sieciowe wtyczki muszą to zapewnić). Dla danego Deploymentu zestaw działających Podów może zmieniać się z chwili na chwilę.

Problem: Jak znaleźć backendy?

Jeśli pewien zestaw Podów (nazwijmy je *backendami*) zapewnia funkcjonalność dla innych Podów (*frontendów*), to pojawia się pytanie: W jaki sposób frontendy mają wiedzieć, z jakim adresem IP się połączyć, skoro backendy się zmieniają? Rozwiązaniem tego problemu jest właśnie **Service**.

Services w Kubernetes

Service API w Kubernetes to abstrakcja, która pomaga udostępniać grupy Podów przez sieć.
Każdy obiekt typu Service definiuje:

- logiczny zestaw punktów końcowych (zwykle są to Pody),
- oraz politykę dostępu, czyli sposób, w jaki te Pody mają być dostępne.

Przykład:

Wyobraź sobie bezstanowy backend przetwarzający obrazy, uruchomiony w trzech replikach.
Wszystkie repliki są równoważne — frontendom nie zależy, z którą z nich się łączą.
Mimo że konkretne Pody backendu mogą się zmieniać, klienci frontendu nie muszą tego wiedzieć ani śledzić zmian.

Service umożliwia takie właśnie oddzielenie (*decoupling*) frontendu od backendu.

Zestaw Podów, do których kieruje ruch Service, jest zwykle określony przez selector zdefiniowany przez użytkownika. Istnieją też inne sposoby określania punktów końcowych usługi — patrz: *Services without selectors*.

Service, Ingress i Gateway

Jeśli Twoja aplikacja komunikuje się przez HTTP, możesz użyć Ingress, aby kontrolować, jak ruch sieciowy dociera do tej aplikacji. Ingress nie jest typem Service, ale pełni rolę punktu wejścia do klastra.
Umożliwia on zdefiniowanie reguł routingu w jednym miejscu, tak aby można było udostępnić wiele komponentów aplikacji działających w klastrze za jednym wspólnym adresem (tzw. listenerem).

Gateway API w Kubernetes oferuje dodatkowe możliwości w porównaniu z Ingress i Service.
Możesz dodać Gateway do swojego klastra — jest to rodzina rozszerzeń API (wdrażanych za pomocą CustomResourceDefinitions) — i dzięki temu precyzyjnie konfigurować dostęp do usług sieciowych działających w klastrze.

Kubernetes

https://cloud.hacktricks.wiki/en/pentesting-cloud/kubernetes-security/index.html

https://kubernetes.io/pl/docs/concepts/overview/

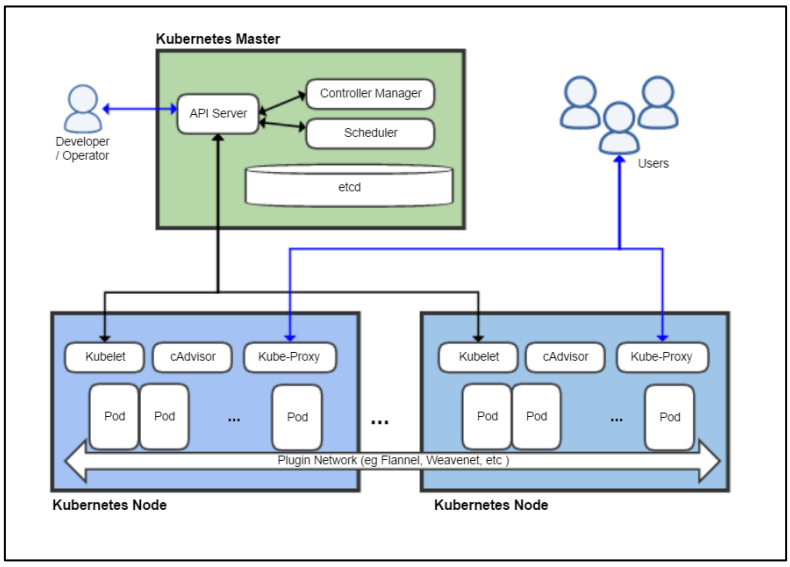
https://en.wikipedia.org/wiki/Kubernetes

Kubernetes znany także jako K8s, to otwartoźródłowy system orkiestracji kontenerów służący do automatyzacji wdrażania oprogramowania, skalowania i zarządzania nim. Oryginalnie zaprojektowany przez Google, projekt jest obecnie utrzymywany przez globalną społeczność kontrybutorów, a znak towarowy należy do Cloud Native Computing Foundation. Kubernetes łączy jeden lub więcej komputerów, będących maszynami wirtualnymi lub fizycznymi (bare metal), w klaster, który może uruchamiać obciążenia w kontenerach. Działa z różnymi środowiskami uruchomieniowymi kontenerów (container runtimes), takimi jak containerd i CRI-O. Jego przydatność w uruchamianiu i zarządzaniu obciążeniami o każdym rozmiarze i charakterze doprowadziła do jego szerokiej adopcji w chmurach i centrach danych. Istnieje wiele dystrybucji tej platformy – zarówno od niezależnych dostawców oprogramowania (ISV), jak i w formie ofert hostowanych w chmurach wszystkich głównych dostawców chmur publicznych.

Oprogramowanie składa się z płaszczyzny sterowania (control plane) oraz węzłów (nodes), na których faktycznie uruchamiane są aplikacje. Zawiera narzędzia takie jak kubeadm i kubectl, które mogą być używane do interakcji z jego API opartym na REST.

Koncepcje

Kubernetes definiuje zestaw klocków konstrukcyjnych („prymitywów"), które łącznie dostarczają mechanizmy wdrażania, utrzymywania i skalowania aplikacji w oparciu o CPU, pamięć lub metryki niestandardowe. Kubernetes jest luźno powiązany (loosely coupled) i rozszerzalny, aby spełniać potrzeby różnych obciążeń. Wewnętrzne komponenty, jak również rozszerzenia i kontenery działające na Kubernetes, polegają na API Kubernetes. Platforma sprawuje kontrolę nad zasobami obliczeniowymi i pamięcią masową poprzez definiowanie zasobów jako obiektów, które następnie mogą być takimi obiektami zarządzane. Kubernetes stosuje architekturę primary/replica (główny/replika). Komponenty Kubernetes można podzielić na te, które zarządzają pojedynczym węzłem, oraz te, które są częścią płaszczyzny sterowania (control plane).



Control plane

Główny węzeł (master node) Kubernetes obsługuje płaszczyznę sterowania klastra Kubernetes, zarządzając jego obciążeniami i kierując komunikacją w całym systemie. Płaszczyzna sterowania Kubernetes składa się z różnych komponentów, takich jak szyfrowanie TLS, RBAC i silna metoda uwierzytelniania, separacja sieciowa – każdy z tych komponentów działa jako osobny proces. Mogą one działać zarówno na pojedynczym węźle głównym, jak i na wielu węzłach głównych (masters), wspierając klastry o wysokiej dostępności. Poszczególne komponenty płaszczyzny sterowania Kubernetes są następujące.

- **Etcd -** etcd to trwały, lekki, rozproszony magazyn danych typu klucz-wartość (oryginalnie opracowany jako część CoreOS). Niezawodnie przechowuje dane konfiguracyjne klastra, reprezentując ogólny stan klastra w dowolnym momencie. etcd preferuje spójność nad dostępnością w przypadku podziału sieciowego (zob. twierdzenie CAP). Spójność jest kluczowa dla prawidłowego planowania (schedulingu) i działania usług.
- **API server -** Serwer API udostępnia API Kubernetes, używając JSON przez HTTP, co zapewnia zarówno wewnętrzny, jak i zewnętrzny interfejs do Kubernetes. Serwer API przetwarza i weryfikuje żądania REST oraz aktualizuje stan obiektów API w etcd, umożliwiając klientom konfigurowanie obciążeń i kontenerów na węzłach roboczych. Serwer API używa mechanizmu watch dostarczanego przez etcd, aby monitorować klaster, wdrażać krytyczne zmiany konfiguracyjne lub przywracać wszelkie odchylenia stanu klastra do stanu pożądanego, zadeklarowanego w etcd. Na przykład operator (człowiek) może określić, że mają działać trzy instancje konkretnego „poda", a etcd zapisuje tę informację. Jeśli kontroler Deployment stwierdzi, że działa tylko dwie instancje (co jest sprzeczne z deklaracją w etcd), planuje utworzenie dodatkowej instancji tego poda.
- **Scheduler -** Scheduler to rozszerzalny komponent, który wybiera węzeł, na którym ma działać nieprzypisany jeszcze pod (pod to podstawowa jednostka obciążenia, która ma być przypisana), bazując na dostępności zasobów i innych ograniczeniach. Scheduler śledzi przydział zasobów na każdym węźle, aby upewnić się, że obciążenie nie jest planowane powyżej dostępnych zasobów. Do tego celu scheduler musi znać wymagania zasobowe, dostępność zasobów oraz inne ograniczenia lub polityki określone przez użytkownika, takie jak jakość usług (quality-of-service), wymagania affinity/anti-affinity oraz lokalizacja danych. Rola scheduler'a polega na dopasowaniu „podaży" zasobów do „popytu" obciążenia. Kubernetes pozwala uruchamiać wiele schedulerów w pojedynczym klastrze. W związku z tym wtyczki schedulerów mogą być tworzone i instalowane jako rozszerzenia działające w procesie natywnego (domyślnego) schedulera, uruchamiane jako oddzielny scheduler, tak długo, jak są zgodne z frameworkiem harmonogramowania Kubernetes. Pozwala to administratorom klastra rozszerzać lub modyfikować zachowanie domyślnego schedulera Kubernetes zgodnie z ich potrzebami.
- **Kontrolery (Controllers) -** Kontroler to pętla rekonsyliacyjna (reconciliation loop), która prowadzi rzeczywisty stan klastra w kierunku stanu pożądanego, komunikując się z serwerem API w celu tworzenia, aktualizacji i usuwania zasobów, którymi zarządza (np. pody lub punkty końcowe usług). Przykładowy kontroler to kontroler ReplicaSet, który obsługuje replikację i skalowanie poprzez uruchamianie określonej liczby kopii poda w klastrze. Kontroler ten zajmuje się także tworzeniem zastępczych podów, jeśli węzeł, na którym działały, ulegnie awarii. Inne kontrolery będące częścią podstawowego systemu Kubernetes obejmują kontroler DaemonSet, uruchamiający dokładnie jeden pod na każdej maszynie (lub na pewnym podzbiorze maszyn), oraz kontroler Job, uruchamiający pody, które mają zakończyć działanie po wykonaniu zadania (np. w ramach zadania wsadowego). Selektory etykiet (label selectors) często stanowią część definicji kontrolera i określają zestaw podów, którymi kontroler zarządza. Controller manager to pojedynczy proces, który zarządza kilkoma podstawowymi kontrolerami Kubernetes (w tym przykładami opisanymi powyżej). Jest on dostarczany jako część standardowej instalacji Kubernetes i reaguje m.in. na utratę węzłów. W klastrze mogą być także instalowane niestandardowe kontrolery (custom controllers), co dodatkowo pozwala rozszerzać zachowanie i API Kubernetes, zwłaszcza w połączeniu z zasobami niestandardowymi (patrz: zasoby niestandardowe, kontrolery i operatorzy poniżej).

Węzły (Nodes)

Węzeł, znany także jako worker lub minion, to maszyna, na której są wdrażane kontenery (obciążenia). Każdy węzeł w klastrze musi uruchamiać środowisko uruchomieniowe kontenerów (container runtime), a także poniższe komponenty, aby umożliwić komunikację i podstawową konfigurację sieciową tych kontenerów.

- **Kubelet -** kubelet jest odpowiedzialny za stan działania każdego węzła, zapewniając, że wszystkie kontenery na węźle są zdrowe. Dba o uruchamianie, zatrzymywanie i utrzymywanie kontenerów aplikacyjnych zorganizowanych w pody, zgodnie ze wskazaniami płaszczyzny sterowania. kubelet monitoruje stan poda i jeśli nie jest on w stanie pożądanym, pod zostaje ponownie wdrożony na tym samym węźle. Status węzła jest przekazywany co kilka sekund poprzez komunikaty heartbeat do serwera API. Gdy płaszczyzna sterowania wykryje awarię węzła, oczekuje się, że kontroler wyższego poziomu zaobserwuje tę zmianę stanu i uruchomi pody na innym zdrowym węźle.
- **Container runtime -** Środowisko uruchomieniowe kontenerów (container runtime) jest odpowiedzialne za cykl życia kontenerów, w tym uruchamianie, doprowadzanie do zgodności (reconciling) oraz usuwanie kontenerów. kubelet komunikuje się z runtime za pomocą Container Runtime Interface (CRI), co oddziela utrzymanie rdzenia Kubernetes od konkretnej implementacji CRI. Początkowo kubelet komunikował się wyłącznie z runtime Docker poprzez tzw. „dockershim". Jednak od listopada 2020 do kwietnia 2022 Kubernetes stopniowo wycofywał ten shim na rzecz bezpośredniej komunikacji z kontenerem poprzez containerd lub zastąpienia Dockera środowiskiem uruchomieniowym zgodnym z CRI. Wraz z wydaniem wersji 1.24 w maju 2022 „dockershim" został całkowicie usunięty. Przykłady popularnych środowisk uruchomieniowych kontenerów kompatybilnych z kubelet obejmują containerd (początkowo obsługiwane poprzez Dockera) i CRI-O.
- **kube-proxy -** kube-proxy to implementacja serwera proxy sieciowego i load balancera. Wspiera on abstrakcję Service wraz z innymi operacjami sieciowymi. Odpowiada za kierowanie ruchu do odpowiedniego kontenera na podstawie adresu IP i numeru portu przychodzącego żądania.

Namespaces

W Kubernetes namespaces są wykorzystywane do podziału zarządzanych zasobów na oddzielne i nieprzenikające się zbiory. Są przeznaczone do użycia w środowiskach z wieloma użytkownikami rozproszonymi między wieloma zespołami lub projektami, a także do separacji środowisk takich jak development, test i produkcja.

Pody (Pods)

Podstawową jednostką planowania (schedulingu) w Kubernetes jest pod, który składa się z jednego lub więcej kontenerów, gwarantująco współlokowanych na tym samym węźle. Każdy pod w Kubernetes otrzymuje unikalny adres IP w obrębie klastra, co pozwala aplikacjom używać portów bez ryzyka konfliktu. Wewnątrz poda wszystkie kontenery mogą się nawzajem odwoływać. Kontener znajduje się wewnątrz poda. Kontener to najniższy poziom mikroserwisu: zawiera działającą aplikację, biblioteki i ich zależności.

Workloads

Kubernetes wspiera kilka abstrakcji obciążeń (workloads), które są wyższego poziomu niż zwykłe pody. Pozwala to użytkownikom deklaratywnie definiować i zarządzać tymi wysokopoziomowymi abstrakcjami zamiast ręcznie zarządzać pojedynczymi podami. Kilka z tych abstrakcji, wspieranych w standardowej instalacji Kubernetes, jest opisanych poniżej.

- **ReplicaSets, ReplicationControllers i Deployments -** Celem ReplicaSet jest utrzymywanie stabilnego zestawu replik podów działających w danym momencie. W związku z tym jest często używany do gwarantowania dostępności określonej liczby identycznych podów. ReplicaSet można uznać za mechanizm grupujący, który pozwala Kubernetes utrzymywać liczbę instancji zadeklarowaną dla danego poda. Definicja ReplicaSet używa selektora, którego ewaluacja identyfikuje wszystkie pody powiązane z daną ReplicaSet. ReplicationController, podobnie jak ReplicaSet, służy temu samemu celowi i zachowuje się podobnie: zapewnia, że zawsze istnieje określona liczba replik poda zgodnie z oczekiwaniami. Obciążenie typu ReplicationController było poprzednikiem ReplicaSet, ale zostało ostatecznie wycofane na rzecz ReplicaSet, aby korzystać z selektorów etykiet opartych na zbiorach. Deployments są mechanizmem wyższego poziomu zarządzania ReplicaSet. Podczas gdy kontroler ReplicaSet zarządza skalą (liczbą replik) ReplicaSet, kontroler Deployment zarządza tym, co się dzieje z ReplicaSet – czy aktualizacja ma być wdrożona, czy wycofana itd. Gdy Deployment jest skalowany w górę lub w dół, skutkuje to zmianą deklarowanego stanu ReplicaSet, a ta zmiana stanu pożądanego jest obsługiwana przez kontroler ReplicaSet.

- **StatefulSets -** StatefulSets to kontrolery, które egzekwują własności unikalności i kolejności pomiędzy instancjami poda i mogą być używane do uruchamiania aplikacji stanowych. Podczas gdy skalowanie aplikacji bezstanowych zwykle oznacza jedynie dodanie większej liczby działających podów, w przypadku obciążeń stanowych jest to trudniejsze, ponieważ stan musi zostać zachowany, jeśli pod zostanie ponownie uruchomiony. Jeśli aplikacja jest skalowana w górę lub w dół, stan może wymagać redystrybucji. Bazy danych są przykładem obciążeń stanowych. Gdy działają w trybie wysokiej dostępności, wiele baz danych ma pojęcie instancji głównej (primary) i instancji wtórnych (secondary). W takim przypadku ważne jest zachowanie kolejności instancji. Inne aplikacje, takie jak Apache Kafka, rozdzielają dane między brokerów; w takim przypadku jeden broker nie jest równoważny drugiemu. Tu ważna jest unikalność instancji.
- **DaemonSets -** DaemonSets są odpowiedzialne za zapewnienie, że pod jest utworzony na każdym pojedynczym węźle w klastrze. Ogólnie rzecz biorąc, większość obciążeń skaluje się w odpowiedzi na pożądaną liczbę replik, zależnie od wymagań dostępności i wydajności aplikacji. Jednak w innych scenariuszach może być konieczne wdrożenie poda na każdym pojedynczym węźle klastra, zwiększając liczbę podów wraz z dodawaniem nowych węzłów i usuwając je, gdy węzły są usuwane. Jest to szczególnie przydatne w przypadkach użycia, gdzie obciążenie ma pewną zależność od konkretnego węzła lub maszyny gospodarza, takich jak zbieranie logów, kontrolery ingress i usługi storage.

Services

Uproszczony widok pokazujący, jak Services współdziałają z siecią Podów w klastrze Kubernetes
Service w Kubernetes to zestaw podów, które współpracują ze sobą, na przykład jedna warstwa aplikacji wielowarstwowej. Zestaw podów, który składa się na Service, jest definiowany przez selektor etykiet. Kubernetes udostępnia dwa tryby odkrywania usług (service discovery): poprzez zmienne środowiskowe oraz poprzez Kubernetes DNS. Service discovery przypisuje stabilny adres IP i nazwę DNS do usługi i równoważy obciążenie w sposób round-robin pomiędzy połączenia sieciowe pod ten adres IP, wśród podów pasujących do selektora (nawet jeśli awarie powodują przenoszenie podów z maszyny na maszynę). Domyślnie Service jest wystawiony wewnątrz klastra (np. pody backend mogą być zgrupowane w Service, a żądania z podów frontend są między nimi równoważone), ale Service może być również wystawiony na zewnątrz klastra (np. aby klienci mogli dotrzeć do podów frontend).

Volumes

Systemy plików w kontenerze Kubernetes domyślnie zapewniają pamięć nietrwałą. Oznacza to, że restart poda wyczyści wszelkie dane na takich kontenerach i dlatego ta forma przechowywania jest mocno ograniczająca we wszystkim poza trywialnymi aplikacjami. Wolumen Kubernetes zapewnia trwałe przechowywanie danych, które istnieje przez cały czas życia samego poda. Takie przechowywanie może być także użyte jako współdzielona przestrzeń dyskowa dla kontenerów wewnątrz poda. Wolumeny są montowane w określonych punktach montowania w systemie plików kontenera, co jest definiowane w konfiguracji poda, i nie mogą montować się na inne wolumeny ani linkować do innych wolumenów. Ten sam wolumen może być zamontowany w różnych miejscach drzewa systemu plików przez różne kontenery.

ConfigMaps i Secrets

Typowym wyzwaniem aplikacyjnym jest decyzja, gdzie przechowywać i jak zarządzać informacją konfiguracyjną, z których część może być wrażliwa. Dane konfiguracyjne mogą być tak drobnoziarniste jak pojedyncze właściwości, albo tak gruboziarniste jak całe pliki konfiguracyjne, np. dokumenty JSON lub XML. Kubernetes dostarcza dwa blisko spokrewnione mechanizmy do rozwiązania tej potrzeby, znane jako ConfigMaps i Secrets. Obydwa pozwalają na wprowadzanie zmian konfiguracyjnych bez konieczności ponownego budowania aplikacji. Dane z ConfigMaps i Secrets będą udostępniane każdej pojedynczej instancji aplikacji, do której te obiekty zostały powiązane za pomocą Deployment. Secret lub ConfigMap jest wysyłany na dany węzeł tylko wtedy, gdy pod na tym węźle go potrzebuje, i jest przechowywany tylko w pamięci tego węzła. Gdy pod zależny od danego Secret lub ConfigMap zostanie usunięty, kopia w pamięci wszystkich powiązanych Secret i ConfigMap jest również usuwana.

Dane z ConfigMap lub Secret są dostępne dla poda w jeden z następujących sposobów:

Jako zmienne środowiskowe, które kubelet przekaże z ConfigMap podczas uruchamiania kontenera; Zamontowane w wolumenie dostępnym w systemie plików kontenera, co wspiera automatyczne przeładowanie bez restartu kontenera. Największa różnica między Secret a ConfigMap polega na tym, że Secret jest specjalnie zaprojektowany do przechowywania danych bezpiecznych i poufnych, chociaż domyślnie nie są one szyfrowane „w spoczynku" (at rest) i wymagają dodatkowej konfiguracji, aby w pełni zabezpieczyć użycie Secret w klastrze. Sekrety są często używane do przechowywania poufnych lub wrażliwych danych, takich jak certyfikaty, poświadczenia do pracy z rejestrami obrazów, hasła i klucze SSH.

Etykiety i selektory (Labels and selectors)

Kubernetes umożliwia klientom (użytkownikom lub wewnętrznym komponentom) dołączanie kluczy zwanych etykietami (labels) do dowolnego obiektu API w systemie, takiego jak pody i węzły. Odpowiadające im selektory etykiet (label selectors) to zapytania przeciwko etykietom, które zwracają pasujące obiekty. Gdy definiuje się Service, można zdefiniować selektory etykiet, które będą używane przez router / load balancer usługi do wyboru instancji podów, do których będzie kierowany ruch. W ten sposób wystarczy zmienić etykiety podów lub zmienić selektory etykiet w usłudze, aby kontrolować, które pody otrzymują ruch, a które nie. Można tego używać do wspierania różnych wzorców wdrożeniowych, takich jak blue–green deployment lub testy A/B. Ta możliwość dynamicznego kontrolowania sposobu, w jaki usługi wykorzystują zasoby implementujące, zapewnia luźne powiązanie w infrastrukturze.

Na przykład, jeśli pody aplikacji mają etykiety dla warstwy systemu (tier) (z wartościami takimi jak frontend, backend) oraz dla ścieżki wydania (release_track) (z wartościami takimi jak canary, production), to operacja na wszystkich podach o tier=backend i release_track=canary może być wykonana poprzez selektor etykiet taki jak:

tier=backend AND release_track=canary

Podobnie jak etykiety, selektory pól (field selectors) także pozwalają wybierać zasoby Kubernetes. W przeciwieństwie do etykiet, selekcja jest oparta na wartościach atrybutów wbudowanych w dany zasób, a nie na kategoryzacji zdefiniowanej przez użytkownika. metadata.name i metadata.namespace są selektorami pól, które będą obecne we wszystkich obiektach Kubernetes. Inne selektory, które mogą być użyte, zależą od typu obiektu / zasobu.

Storage

Kontenery pojawiły się jako sposób na przenośność oprogramowania. Kontener zawiera wszystkie pakiety potrzebne do uruchomienia usługi. Dostarczany system plików sprawia, że kontenery są wyjątkowo przenośne i łatwe w użyciu w środowiskach deweloperskich. Kontener może zostać przeniesiony ze środowiska deweloperskiego do testowego lub produkcyjnego bez żadnych lub przy minimalnych zmianach konfiguracji. Historycznie Kubernetes nadawał się tylko do usług bezstanowych. Jednak wiele aplikacji ma bazę danych, która wymaga trwałości, co doprowadziło do stworzenia trwałego storage dla Kubernetes. Implementacja trwałego storage dla kontenerów jest jednym z głównych wyzwań administratorów Kubernetes, zespołów DevOps i inżynierów chmurowych. Kontenery mogą być efemeryczne, ale coraz więcej danych nimi zarządzanych już takie nie jest, więc trzeba zapewnić przetrwanie danych w przypadku zakończenia działania kontenera lub awarii sprzętu. Podczas wdrażania kontenerów przy użyciu Kubernetes lub aplikacji skonteneryzowanych organizacje często zdają sobie sprawę, że potrzebują trwałego storage. Muszą dostarczyć szybkie i niezawodne storage dla baz danych, obrazów root i innych danych używanych przez kontenery. Oprócz ogólnego krajobrazu, Cloud Native Computing Foundation (CNCF) opublikowała inne informacje na temat trwałego storage w Kubernetes, w tym wpis na blogu pomagający zdefiniować wzorzec container attached storage. Ten wzorzec można rozumieć jako taki, który używa samego Kubernetes jako komponentu systemu lub usługi storage. Więcej informacji o względnej popularności tych i innych podejść można znaleźć w przeglądzie krajobrazu CNCF, który pokazał, że OpenEBS – platforma Stateful Persistent Storage od Datacore Software – oraz Rook – projekt orkiestracji storage – to dwa projekty najczęściej będące w fazie ewaluacji jesienią 2019 roku. Container Attached Storage to rodzaj przechowywania danych, który pojawił się wraz ze wzrostem znaczenia Kubernetes. Podejście lub wzorzec Container Attached Storage polega na tym, że sam Kubernetes jest wykorzystywany do pewnych możliwości, przy jednoczesnym dostarczaniu głównie blokowego, plikowego, obiektowego storage oraz interfejsów do obciążeń działających na Kubernetes. Typowe cechy Container Attached Storage obejmują użycie rozszerzeń Kubernetes, takich jak Custom Resource Definitions, oraz użycie samego Kubernetes do funkcji, które inaczej byłyby osobno rozwijane i wdrażane dla systemu storage lub zarządzania danymi. Przykładami funkcjonalności dostarczanych przez Custom Resource Definitions lub przez sam Kubernetes są logika ponawiania (retry logic), dostarczana przez Kubernetes, oraz tworzenie i utrzymywanie inwentarza dostępnych nośników storage i wolumenów, zwykle dostarczane poprzez Custom Resource Definition.

Container Storage Interface (CSI)

W wersji Kubernetes 1.9 wprowadzono początkowe wydanie w fazie Alpha interfejsu Container Storage Interface (CSI). Wcześniej wtyczki wolumenów storage były dołączane do dystrybucji Kubernetes. Poprzez stworzenie ustandaryzowanego CSI kod wymagany do integracji z zewnętrznymi systemami storage został oddzielony od bazowego kodu Kubernetes. Zaledwie rok później funkcja CSI uzyskała status GA (General Availability) w Kubernetes.

API

Kluczowym komponentem płaszczyzny sterowania Kubernetes jest serwer API, który udostępnia API HTTP, mogące być wywoływane przez inne części klastra, a także przez użytkowników końcowych i komponenty zewnętrzne. Jest to API typu REST i ma charakter deklaratywny. To jest to samo API, które jest udostępniane płaszczyźnie sterowania. Serwer API jest wspierany przez etcd, który przechowuje wszystkie rekordy w sposób trwały.

https://kubernetes.io/docs/concepts/overview/kubernetes-api/

Kubernetes Security

- https://kubernetes.io/docs/tasks/configure-pod-container/security-context/
- https://kubernetes.io/docs/concepts/security/pod-security-standards/
- https://kubernetes.io/docs/concepts/security/
- https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html
- https://cloud.hacktricks.wiki/en/pentesting-cloud/kubernetes-security/index.html
- https://medium.com/marionete/kubernetes-securitycontext-with-practical-examples-67d890558d11

Helm

- https://www.baeldung.com/ops/kubernetes-helm
- https://www.datacamp.com/tutorial/helm-chart
- https://www.freecodecamp.org/news/what-is-a-helm-chart-tutorial-for-kubernetes-beginners/

Helm automatyzuje tworzenie, pakowanie, konfigurowanie i wdrażanie aplikacji Kubernetes, łącząc pliki konfiguracyjne w jeden, możliwy do ponownego użycia pakiet. Automatyzacja tej pracy pomaga deweloperom pracującym z mikroserwisami. W architekturze mikroserwisów tworzysz coraz więcej mikroserwisów w miarę rozwoju aplikacji, co sprawia, że zarządzanie staje się coraz trudniejsze. Kubernetes, otwartoźródłowa technologia orkiestracji kontenerów, upraszcza ten proces, grupując wiele mikroserwisów w jedno wdrożenie. Jednak zarządzanie aplikacjami Kubernetes w całym cyklu życia rozwoju niesie ze sobą własny zestaw wyzwań, w tym zarządzanie wersjami, przydział zasobów, aktualizacje i wycofywanie zmian. Helm zapewnia jedno z najbardziej przystępnych rozwiązań tego problemu, czyniąc wdrożenia bardziej spójnymi, powtarzalnymi i niezawodnymi.

Uproszczenie zarządzania Kubernetes za pomocą Helm

Kubernetes wdraża aplikacje przy użyciu plików konfiguracyjnych YAML. W przypadku złożonych wdrożeń z częstymi aktualizacjami trudno jest śledzić wszystkie różne wersje tych plików. Helm to przydatne narzędzie, które utrzymuje pojedynczy plik YAML wdrożenia z informacjami o wersji. Ten plik pozwala skonfigurować i zarządzać złożonym klastrem Kubernetes za pomocą kilku poleceń. W kolejnych sekcjach omówimy, jak komponenty Helm upraszczają zarządzanie Kubernetes.

Helm Chart

Helm Chart to pakiet, który zawiera wszystkie zasoby potrzebne do wdrożenia aplikacji w klastrze Kubernetes. Ten pakiet zawiera pliki YAML dla wdrożeń, usług, sekretów i map konfiguracyjnych, które konfigurują aplikację zgodnie z potrzebami.

Helm Chart łączy pliki YAML i szablony w celu utworzenia plików konfiguracyjnych na podstawie określonych parametrów. Możesz dostosowywać te pliki konfiguracyjne do różnych środowisk i tworzyć konfiguracje, które można ponownie wykorzystać w wielu wdrożeniach. Dodatkowo możesz wersjonować i zarządzać każdym Helm chart indywidualnie, co ułatwia utrzymanie wielu wersji aplikacji z różnymi konfiguracjami.

Konfiguracje

Konfiguracja zawiera ustawienia aplikacji, które zwykle przechowujesz w pliku YAML. Klaster Kubernetes wdraża zasoby na podstawie wartości konfiguracji.

Releases

Release to działająca instancja Helm chart. Gdy uruchamiasz polecenie helm install, pobiera ono pliki konfiguracyjne i helm charts oraz wdraża wszystkie zasoby Kubernetes.

Architektura

Architektura ma dwa główne komponenty: klienta i bibliotekę.

Klient Helm pomaga użytkownikom kontrolować rozwój helm charts, zarządzać repozytoriami i wydawać oprogramowanie za pomocą wiersza poleceń. Podobnie jak w przypadku korzystania z klienta bazy danych MySQL do uruchamiania poleceń MySQL, używasz klienta Helm do uruchamiania poleceń Helm.

Biblioteka Helm wykonuje całą ciężką pracę. Zawiera rzeczywisty kod do wykonywania operacji określonych w poleceniu Helm. Biblioteka Helm obsługuje połączenie plików konfiguracyjnych i helm charts w celu utworzenia dowolnego wydania.

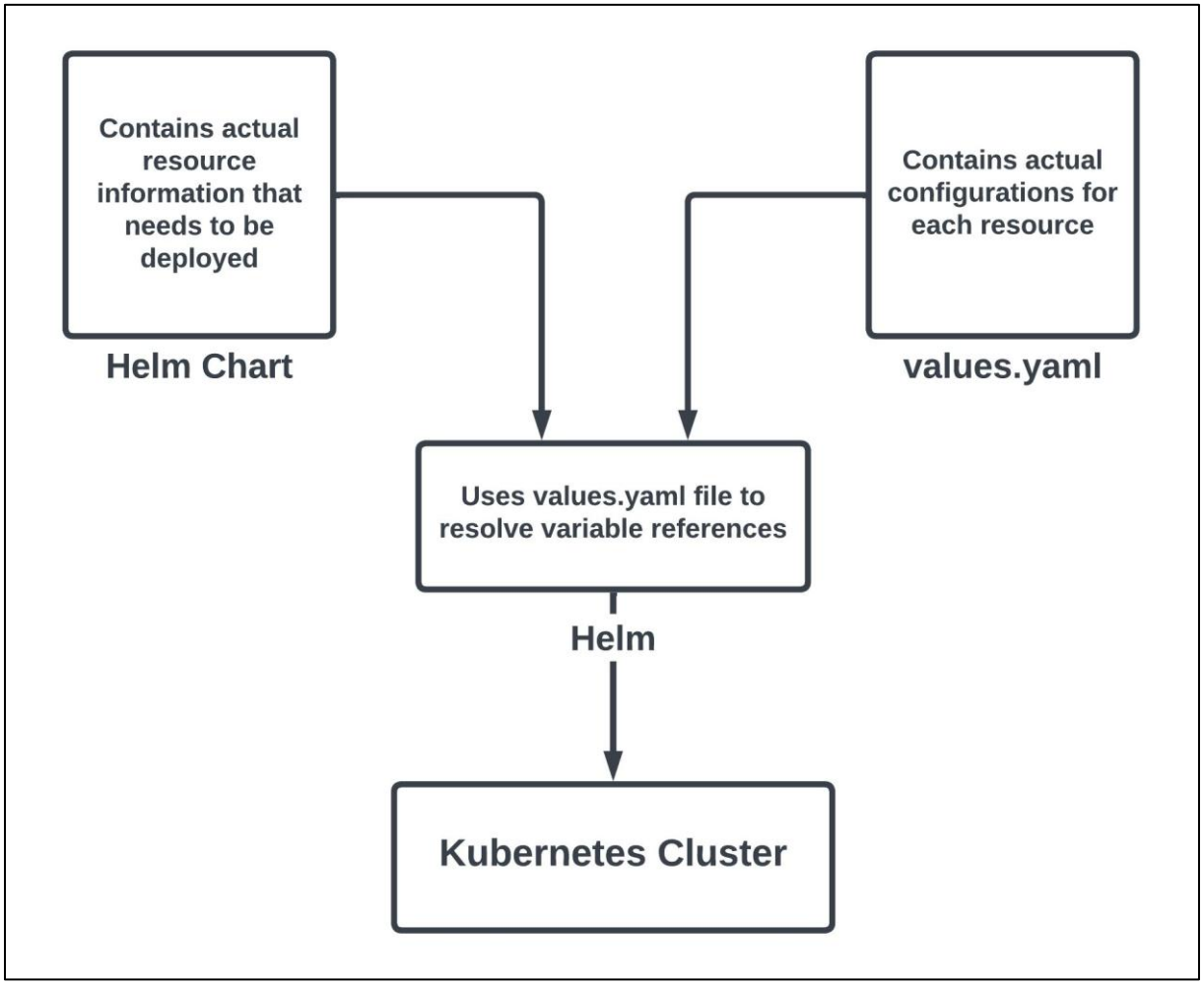
Architektura Helm została znacznie ulepszona między wersjami 2 i 3. W wersji 2 istniał serwer Tiller, który łączył klienta Helm z serwerem API Kubernetes. Śledził on wszystkie zasoby utworzone przy użyciu Helm.

W wersji 3 serwer Tiller został usunięty. Teraz używa bezpośredniego połączenia API w architekturze opartej wyłącznie na kliencie. Umożliwia to interakcję z serwerem API Kubernetes.

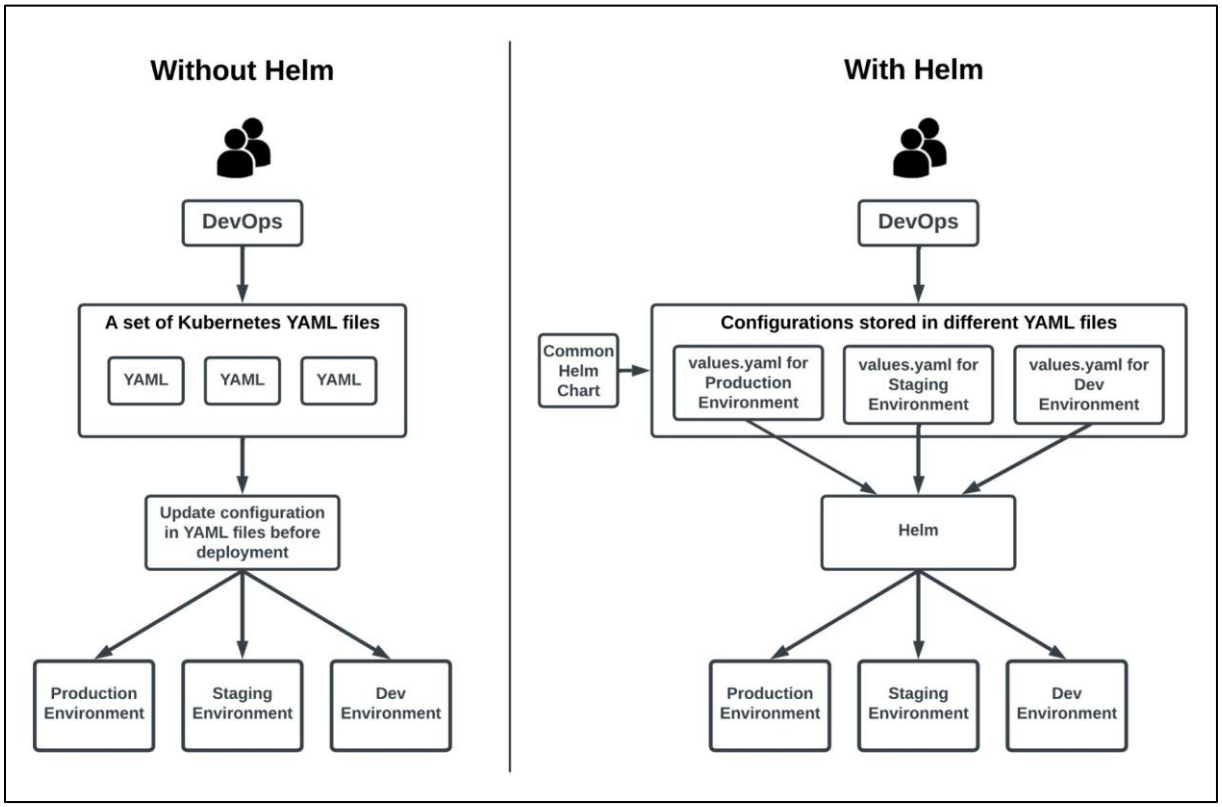
Jak działa Helm

Biblioteka aplikacji Helm używa helm chartów do definiowania, tworzenia, instalowania i aktualizowania aplikacji Kubernetes. Możesz używać Helm charts do zarządzania manifestami Kubernetes bez użycia interfejsu wiersza poleceń Kubernetes (CLI). Nie musisz pamiętać skomplikowanych poleceń Kubernetes, aby kontrolować klaster.

Rozważmy praktyczny scenariusz. Załóżmy, że chcemy wdrożyć swoją aplikację w środowisku produkcyjnym z dziesięcioma replikami. Określamy to w pliku YAML wdrożenia aplikacji i uruchamiamy wdrożenie za pomocą polecenia kubectl. Teraz uruchamiamy tę samą aplikację w środowisku testowym. Załóżmy, że potrzebujemy trzech replik w testowym i że uruchomisz wewnętrzną kompilację aplikacji w tym środowisku. Aby zaktualizować, zmieniamy liczbę replik i tag obrazu Dockera w pliku YAML wdrożenia. Po wprowadzeniu tych zmian zastosujemy plik do testowego klastra Kubernetes. W miarę jak nasza aplikacja staje się bardziej złożona, liczba plików YAML rośnie. Ostatecznie rośnie również liczba konfigurowalnych pól w pliku YAML. Wkrótce aktualizowanie wielu plików YAML w celu wdrożenia tej samej aplikacji w różnych środowiskach staje się trudne do zarządzania Używając Helm, możemy parametryzować pola w zależności od środowiska. Zamiast używać stałej wartości dla replik i obrazów Dockera, możesz pobierać te wartości z innego pliku. Ten plik nazywa się values.yaml.



Teraz możemy utrzymywać plik wartości dla każdego środowiska z odpowiednimi wartościami dla każdego z nich. Helm pomaga oddzielić konfigurowalne wartości pól od właściwej konfiguracji YAML.



Helm repositories

Repozytorium Helm to miejsce, w którym można przesyłać Helm chart. Można także utworzyć prywatne repozytorium, aby udostępniać charty w swojej organizacji. Artifact Hub to globalne repozytorium Helm, które zawiera różnorodne helm charty z możliwością wyszukiwania. Artifact Hub robi dla chartów Helm to, co Docker Hub dla obrazów Dockera.

Używanie Helm do wdrożeń i wycofywania zmian

Po utworzeniu Helm chart wdrożenie aplikacji jest łatwe. Aby wdrożyć wszystkie zasoby Kubernetes, trzeba uruchomić tylko kilka poleceń Helm.

Wyobraź sobie, że Twoja aplikacja działa w klastrze Kubernetes wdrożonym za pomocą Helm chart. Teraz chcesz skonfigurować rozwiązanie do monitorowania, takie jak Prometheus, dla swojej aplikacji. Masz dwie opcje:

1. Utworzyć wszystkie wdrożenia i usługi dla aplikacji Prometheus od podstaw. To czasochłonne zadanie.
2. Wyszukać Prometheus chart w Artifact Hub, zaktualizować konfigurację zgodnie z wymaganiami i zainstalować go. Proces ten jest podobny do używania istniejących obrazów Dockera z Docker Hub w instrukcji FROM w pliku Dockerfile.

Możesz zainstalować dowolny Helm chart, wykonując następujące kroki:

- Zarejestruj repozytorium Helm lokalnie, aby pomóc bibliotece Helm zidentyfikować lokalizację, z której ma pobrać Helm chart.
- Pobierz wszystkie Helm chart dostępne w skonfigurowanym repozytorium.
- Zaktualizuj plik values.yaml (jeśli to konieczne) i zainstaluj określony Helm chart za pomocą polecenia helm install.

Kolejną zaletą korzystania z Helm jest to, że ułatwia proces wycofywania zmian. Helm działa na poziomie aplikacji, co oznacza, że utrzymuje stan działającej aplikacji, znany również jako wydanie. Załóżmy, że wdrożyłeś nową wersję aplikacji, która nie działa zgodnie z oczekiwaniami. Używając polecenia helm rollback, możesz powrócić do poprzedniego stabilnego wydania. To polecenie cofa wszystkie wdrożenia, usługi i zasoby Kubernetes. Bez Helm wycofywanie zmian byłoby trudne. Musiałbyś wykonać wycofanie dla każdego zasobu Kubernetes, co mogłoby utrudnić zarządzanie złożonymi aplikacjami.

Helm i CI/CD

Helm jest ważny, ponieważ ułatwia budowanie i testowanie aplikacji Kubernetes w CI/CD. Może automatycznie wdrażać aplikację w dowolnym środowisku i skracać czas kompilacji. Aby uzyskać większą elastyczność w środowiskach budowania i testowania, możesz użyć samodzielnie hostowanego runnera dla swoich pipeline.

Zalety i wady korzystania z Helm

Niektóre scenariusze korzystają z Helm, podczas gdy inne przypadki użycia nie są tak dobrze do tego przystosowane. Ta sekcja omawia, kiedy Helm jest przydatny, a kiedy nie. Opisuje również oznaki, że Twoja organizacja może skorzystać z używania Helm.

Kiedy używać Helm

Helm jest pomocny, gdy Twój projekt używa Kubernetes do uruchamiania złożonych aplikacji z wieloma mikroserwisami. Dzięki Helm możesz automatyzować wdrażanie i zarządzanie aplikacjami, zmniejszając pracę ręczną i poprawiając niezawodność oraz stabilność systemu. Helm daje dostęp do wielu prekonfigurowanych pakietów, co ułatwia dodawanie nowych funkcji i możliwości do aplikacji.

Helm pomaga zarządzać komponentami aplikacji, organizując je w charty dla łatwej instalacji i aktualizacji. Zmniejsza ilość pracy ręcznej potrzebnej do utrzymania aplikacji. Pomaga również zapobiegać błędom i niespójnościom, które mogą wystąpić podczas ręcznego zarządzania złożonymi systemami.

Helm pomaga wdrażać kontenery w różnych środowiskach, takich jak development, staging i produkcja. Ułatwia zarządzanie kontenerami podczas rozwoju.

Kiedy Helm nie sprawdza się najlepiej

Helm jest najbardziej pomocny w zarządzaniu wieloma kontenerami jednocześnie. Helm nie działa tak dobrze w projektach, które wymagają wdrożenia pojedynczego kontenera na serwerze. W takim przypadku użycie Helm może nie być konieczne i może dodać złożoności do procesu. Jeśli masz tylko kilka aplikacji Kubernetes, prawdopodobnie możesz nimi zarządzać ręcznie, bez menedżera pakietów. Używanie Helm może nie przynieść znaczących korzyści.

Kiedy powinieneo korzystać się z Helm

Istnieje kilka wskaźników, że projekt może skorzystać z użycia Helm. Na przykład rozważ projekt, który obejmuje wiele aplikacji Kubernetes wymagających zarządzania i wdrażania jako spójnej całości. Helm ułatwia zarządzanie i wdrażanie aplikacji, pakując je w jeden chart dla wygody. Jeśli często aktualizujesz i wdrażasz aplikacje Kubernetes, narzędzia Helm mogą pomóc w zarządzaniu cyklem życia aplikacji, w tym w wycofywaniu zmian. Jeśli Twój projekt obejmuje wiele zespołów lub osób współpracujących przy aplikacjach Kubernetes, Helm może pomóc. Możesz udostępniać i śledzić różne wersje chartów, pomagając zespołom współpracować i zachować spójność podczas wdrażania.

Materiały dodatkowe:

- Od czego zacząć | Kubernetes
- https://github.com/kubernetes/kubernetes
- https://github.com/collabnix/kubetools
- https://collabnix.com/kubectl-cheatsheet/
- GitHub - k3s-io/k3s: Lightweight Kubernetes
- https://www.youtube.com/playlist?list=PLy7NrYWoggjziYQIDorlXjTvvwweTYoNC
- https://www.youtube.com/watch?v=s_o8dwzRlu4&t=1s
- https://www.youtube.com/watch?v=X48VuDVv0do&t=1s
- https://www.youtube.com/watch?v=fy8SHvNZGeE
- https://www.youtube.com/watch?v=jUYNS90nq8U
- https://www.youtube.com/watch?v=w51lDVuRWuk
- https://spacelift.io/blog/kubernetes-tutorial
- https://jayvardhanchandel.medium.com/intro-to-kubernetes-tryhackme-thm-write-up-walkthrough-b4ee5d4e96e0
- https://www.freecodecamp.org/news/learn-kubernetes-handbook-devs-startups-businesses/
- https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/
- https://kubernetes.io/docs/concepts/workloads/controllers/job/
- https://github.com/collabnix/kubelabs (Local kubernetes workshop)
- https://github.com/kelseyhightower/kubernetes-the-hard-way (Local kubernetes workshop)

Przykłady

Instalacja Kubernetes na Ubuntu:

- https://www.geeksforgeeks.org/installation-guide/how-to-install-and-configure-kubernetes-on-ubuntu/
- https://kubernetes.io/docs/tasks/tools/

Jak używać kubernetes:

- https://kubernetes.io/docs/tutorials/hello-minikube/
- https://mosesmbadi.medium.com/kubernetes-but-locally-aa5fbd671763
- https://kubernetes.io/docs/tasks/

Przykłady użycia helm charts:

- https://medium.com/htc-research-engineering-blog/a-simple-example-for-helm-chart-fbb5c7208e94
- https://www.datacamp.com/tutorial/helm-chart
- https://helm.sh/docs/topics/charts/

Zadania praktyczne

Zadanie 1 - Kubernetes playground

- Uruchom laboratorium Kubernetes na stronie Killercoda.
- Wykonaj dostępne w nim ćwiczenia dotyczące podstaw Kubernetes.
- Sprawdź działanie podstawowych komend kubectl, tworzenie podów, deploymentów i usług.
- Upewnij się, że potrafisz utworzyć, skalować i usunąć aplikację w klastrze.
    - Killercoda Interactive Environments
        - https://training.play-with-kubernetes.com/kubernetes-workshop/
        - https://labs.play-with-k8s.com/#
    - https://sharelearn.net/practice/k8slabs/
    - https://kodekloud.com/free-labs/kubernetes (Alternatywnie, lecz zalecam skorzystanie z pierwszego linku)

Zadanie 2 - Cluster security and hardening

- Uruchom wskazane pokoje TryHackMe dotyczące bezpieczeństwa Kubernetes.
- Wykonaj wszystkie zadania krok po kroku według instrukcji w laboratoriach.
- Zidentyfikuj słabe punkty konfiguracji klastra i zapisz proponowane działania naprawcze.
- Zwróć uwagę na zasady RBAC, uprawnienia kont serwisowych oraz bezpieczeństwo runtime.
    - https://tryhackme.com/room/k8sbestsecuritypractices
    - https://tryhackme.com/room/clusterhardening
    - https://tryhackme.com/room/k8sruntimesecurity

Zadanie 3 - Microservices Architectures

- Uruchom lab dotyczący architektury mikroserwisów.
- Przeanalizuj sposób komunikacji między usługami w środowisku Kubernetes.
- Wykonaj zadania z laboratorium i sprawdź, jak mikroserwisy reagują na awarie innych komponentów.
    - https://tryhackme.com/room/microservicearchitectures

Zadanie 4 – Kubernetes hacking CTF

- Uruchom wybrany pokój CTF na TryHackMe.
- Wykonaj wszystkie zadania polegające na analizie, eskalacji uprawnień i przejmowaniu zasobów klastra.
- Zanotuj znalezione flagi oraz zastosowane techniki ataku i obrony.
- Po zakończeniu przeanalizuj, jakie błędy konfiguracyjne umożliwiły atak.
    - https://tryhackme.com/room/insekube
    - https://tryhackme.com/room/kubernetesforyouly

Zadanie 5 – HELM – Dodatkowe (Do wykonania lokalnie)

- https://helm.sh/docs/chart_template_guide/getting_started/
- https://helm.sh/docs/intro/quickstart/

1. Uruchom środowisko z dostępem do klastra Kubernetes (np. z laboratorium KodeKloud lub Minikube).
2. Zainstaluj Helm (jeśli nie jest dostępny).
3. Sprawdź poprawność instalacji poleceniem:

```
helm version
```

4. Dodaj oficjalne repozytorium Helm:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

5. Zainstaluj przykładową aplikację nginx z repozytorium Bitnami:

```
helm install my-nginx bitnami/nginx
```

6. Sprawdź, czy aplikacja została poprawnie wdrożona:

```
kubectl get all
```

7. Usuń wdrożenie po zakończeniu testu:

```
helm uninstall my-nginx
```

Zadanie 6 – Intro to Pipeline automation - Dla chętnych

- Uruchom laboratorium dotyczące automatyzacji pipeline'u CI/CD.
- Wykonaj zadania opisane w pokoju - od budowy obrazu po automatyczne wdrożenie.
    - https://tryhackme.com/room/introtopipelineautomation

Polecenia do Sprawozdania

1. Opisz cel i zakres przeprowadzonych laboratoriów - Wyjaśnij, do czego można wykorzystać rozwiązania takie jak *Ansible, Terraform, Docker, Kubernetes, mikroserwisy, Helm* oraz jakie problemy rozwiązują w środowiskach IT.
2. Przedstaw zalety i wady środowisk rozproszonych oraz porównaj je do aplikacji monolitycznych.
3. Omów podstawowe mechanizmy działania Ansible (Uwzględnij sposób działania playbooków, ról, modułów oraz proces automatyzacji konfiguracji)
4. Omów podstawowe mechanizmy działania Docker (Opisz pojęcia obrazu, kontenera, warstw, sieci i wolumenów)
5. Omów podstawowe mechanizmy działania Kubernetes oraz sposoby zarządzania aplikacjami w klastrze. (Uwzględnij rolę podów, deploymentów, usług, przestrzeni nazw i skalowania)
6. Wymień i opisz metody hardeningu klastra Kubernetes lub kontynerów Docker (Przedstaw najważniejsze zasady bezpieczeństwa, konfiguracje i dobre praktyki.)
7. Przeanalizuj zagrożenia bezpieczeństwa w środowiskach kontenerowych i rozproszonych oraz sposoby ich wykrywania.
8. Omów potencjalne zagrożenia bezpieczeństwa podczas zarządzania serwerem aplikacyjnym oraz sposoby ich wykrywania i przeciwdziałania.