ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
СПЕЦИАЛНОСТ КОМПЮТЪРНО И СОФТУЕРНО ИНЖЕНЕРСТВО

ОПЕРАЦИОННИ СИСТЕМИ

Студент: Валери Ивайлов Райков

Фак. №: 121222139

Група: 41А

Курсов проект на тема:

4. DOCKER ТЕХНОЛОГИЯ И KUBERNETES - АНАЛИЗ И ВРЪЗКА МЕЖДУ ТЕХНОЛОГИИТЕ. ПРИМЕР.

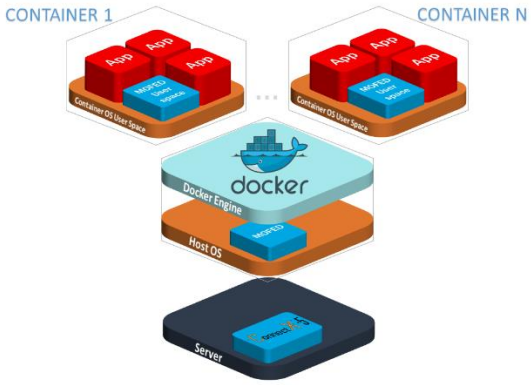
Дата на предаване: 26/11/2024г.

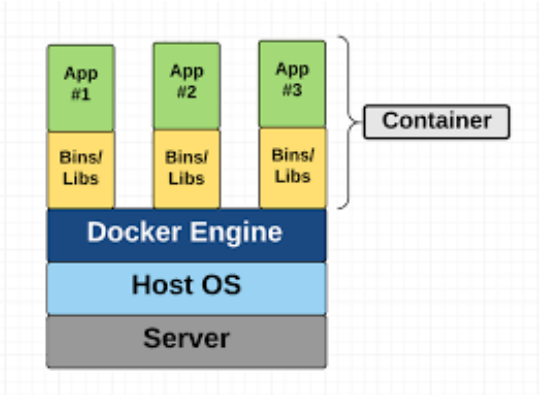
Съдържание на Курсовия проект

Цел на проекта и основни въпроси, които ще разгледа

Как Docker и Kubernetes помагат за създаването и управлението на контейнери. Анализ на двете технологии, връзката между тях и пример за тяхното практическо приложение.

1. Въведение





- **Значимост на контейнеризацията в съвременния софтуерен свят:**

Контейнеризацията се е утвърдила като основен инструмент в съвременното софтуерно развитие, благодарение на способността си да осигури преносимост, ефективност и мащабируемост. В много съвременни приложения, особено тези с микросървисна архитектура, контейнеризацията предоставя рамка за изолиране на приложенията и техните зависимости (код, библиотеки, конфигурации и т.н.) в самостоятелни и преносими среди – контейнери. Това позволява на разработчиците и системните администратори да изграждат, разгръщат и управляват приложенията по-бързо и по-надеждно в различни среди и на различни платформи.

- **Какво представлява контейнеризацията и защо е ключова за операционните системи и DevOps:**

Контейнеризацията е метод за виртуализация, който създава изолирани и независими среди, наречени контейнери, за да стартира софтуерни приложения. В тези среди са включени всички зависимости на приложението – библиотеки, конфигурации и файлове, нужни за коректното му функциониране. Контейнерите са напълно изолирани един от друг и използват споделено ядро на хост операционната система. Това ги прави значително по-леки от виртуалните машини (VM) и им позволява бързо стартиране и мигриране между различни среди.

Ето някои ключови характеристики на контейнеризацията за операционните системи и DevOps:

1. **Изолация и консистентност между среди:**

- Контейнеризацията осигурява пълна изолация на приложенията и техните зависимости, което предотвратява конфликти между различните среди (работна, тестова, продукционна) и подсигурява, че приложението ще функционира по същия начин навсякъде (Платформена независимост). Това е особено важно за сложни софтуерни среди, където различни приложения могат да изискват различни версии на библиотеки и конфигурации. С контейнерите тези зависимости са отделени за всяко приложение и се съхраняват в неговия контейнер.

2. **Ефективност на ресурсите:**

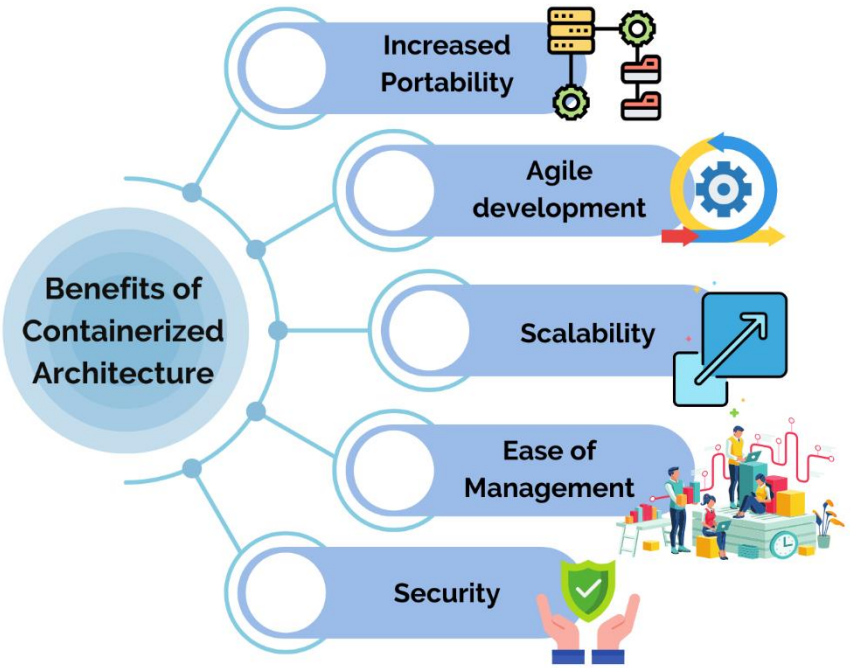
- За разлика от виртуалните машини, контейнерите не изискват цяла операционна система за всяко приложение и могат да споделят ядрото на хост системата. Това ги прави значително по-леки, като позволяват по-ефективна употреба на ресурси и бързо мащабиране на приложенията. Лекотата на контейнерите позволява стартиране и спиране на различни приложения за секунди, което е от съществено значение за динамичните и често променящи се среди.

3. **Подкрепа на DevOps и CI/CD:**

- Контейнеризацията е в основата на DevOps практиките, тъй като значително опростява процесите по непрекъсната интеграция и доставяне (CI/CD). Разработчиците могат да създават контейнерни изображения, които съдържат всичко необходимо за стартирането на приложението, и да ги качват в регистри като Docker Hub. Контейнерите позволяват лесно и бързо прехвърляне на приложения между разработчиците и системните администратори. Това ускорява целия процес на разработка, тестване и внедряване, като същевременно гарантира стабилност и консистентност на приложенията.

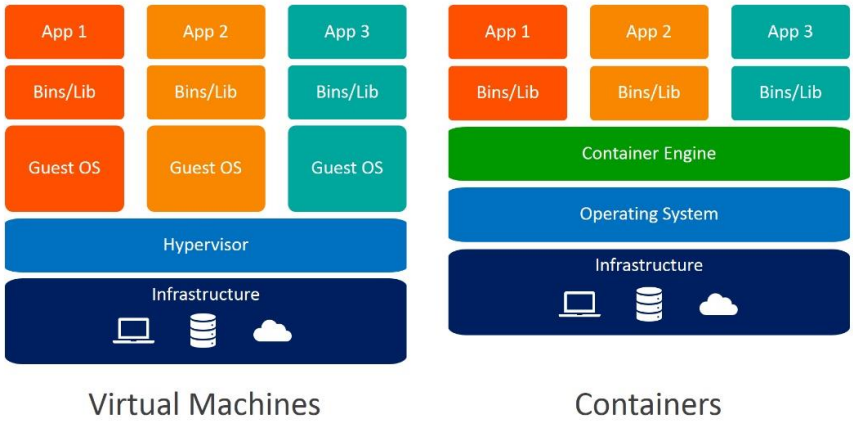
4. **Подходящи за микросървисна архитектура:**

- Контейнеризацията се комбинира добре с микросървисната архитектура, при която всяка част от приложението е отделен модул или услуга, която може да бъде развивана, тествана и мащабирана самостоятелно.



- **Кратка история на контейнерите и как те се отличават от виртуалните машини**

Концепцията за изолация на процесите е въведена преди няколко десетилетия (1970-80-те), когато операционни системи като UNIX започват да експериментират с начини за отделяне на процеси и ресурси, за да подобрят сигурността и ефективността на многопотребителските системи. През 1979г. UNIX въвежда командата **chroot**, която позволява създаването на нов коренов каталог за процеси. Това поставя началото на технологията за изолация, тъй като chroot позволява процесите да бъдат ограничени до специфична част от файловата система. През 2000г. FreeBSD развива концепцията, като създава **Jails** – технология за изолация, която разделя файловата система, мрежата и потребителите. Това позволява по-голяма гъвкавост и сигурност в UNIX-базираните системи и се смята за едно от първите истински решения за изолация на процеси. През 2008г. се появява **LXC (Linux Containers)** - първият проект, който предлага пълна контейнерна изолация на процеси в Linux, като използва функции на ядрото като **cgroups** и **namespaces**. Това позволява създаването на изцяло изолирани среди в Linux. **Docker** и съвременната контейнеризация се появява през 2013г. като първата масово достъпна платформа за контейнеризация. Той значително опростява процеса на създаване и управление на контейнери, като въвежда концепцията за контейнерни **изображения (images)** и **регистри (registry)**, което улеснява споделянето и управлението на приложения в контейнери. Оттогава Docker непрекъснато се развива и се утвърждава като една от най-ключовите технологии в сферата на операционните системи и DevOps.



- **Разлики между контейнери и виртуални машини (VM)**

В тази част от реферата ще представя основните разлики между контейнерите и виртуалните машини, без да се спирам на детайлни обяснения за виртуалните машини, тъй като темата е много обширна и принадлежи на друг курсов проект.

Контейнерите и виртуалните машини се използват за виртуализация, но има ключови различия между тях, които определят предимствата на контейнерите:

Основни различия между контейнери и виртуални машини:

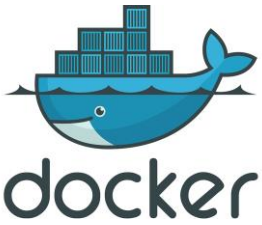
1. **Тегло и ефективност**:
    - Контейнерите не съдържат пълна операционна система и използват ядрото на хост системата, което ги прави значително по-леки. Те заемат по-малко ресурси и позволяват стартиране за части от секундата. Виртуалните машини, от своя страна, изискват пълно копие на операционната система, което ги прави по-големи и по-бавни за стартиране.
2. **Изолация и сигурност**:
    - Виртуалните машини осигуряват по-високо ниво на изолация, тъй като всяка VM има собствена операционна система и е напълно независима от другите. Това ги прави по-сигурни, но значително увеличава изразходваните ресурси. Контейнерите, макар и изолирани, споделят ядрото на хост операционната

система, което може да ги направи уязвими при некоректна конфигурация на сигурността.

3. **Скорост на стартиране и преносимост**:
    - Контейнерите могат да се стартират почти мигновено, тъй като са изградени около изолирани процеси в същата ОС. Виртуалните машини, от друга страна, изискват време за стартиране на пълна операционна система, което ги прави по-бавни. Контейнерите са по-преносими, тъй като са изградени в съответствие със стандарти като Open Container Initiative (OCI) и могат лесно да бъдат преместени между различни среди, докато виртуалните машини са обвързани с хипервайзъра, на който са създадени.
4. **Мащабиране и управление**:
    - Контейнерите се мащабират бързо и лесно благодарение на инструменти като Kubernetes, които автоматизират разгръщането и управлението на клъстери от контейнери. Виртуалните машини също могат да бъдат мащабирани, но изискват много повече ресурси и инфраструктура.

Virtual Machine	*Container*
• Heavyweight	• Lightweight
• Limited performance	• Native performance
• Each VM runs in its own OS	• All containers share the host OS
• Hardware-level virtualization	• OS virtualization
• Startup time in minutes	• Startup time in milliseconds
• Allocates required memory	• Requires less memory space
• Fully isolated and hence more secure	• Process-level isolation, possibly less secure

2. Docker технология



Docker е софтуерна платформа, която предлага стандартизиран метод за изграждане и разпространение на приложения чрез леки, изолирани контейнери. Тези контейнери съдържат всички необходими зависимости, конфигурации и библиотеки, които позволяват на приложението да функционира по идентичен начин в различни среди - от разработка и тестване до продукция. Контейнерите могат да комуникират помежду си по строго дефинирани канали. Услугата е безплатна, но предлага и платена

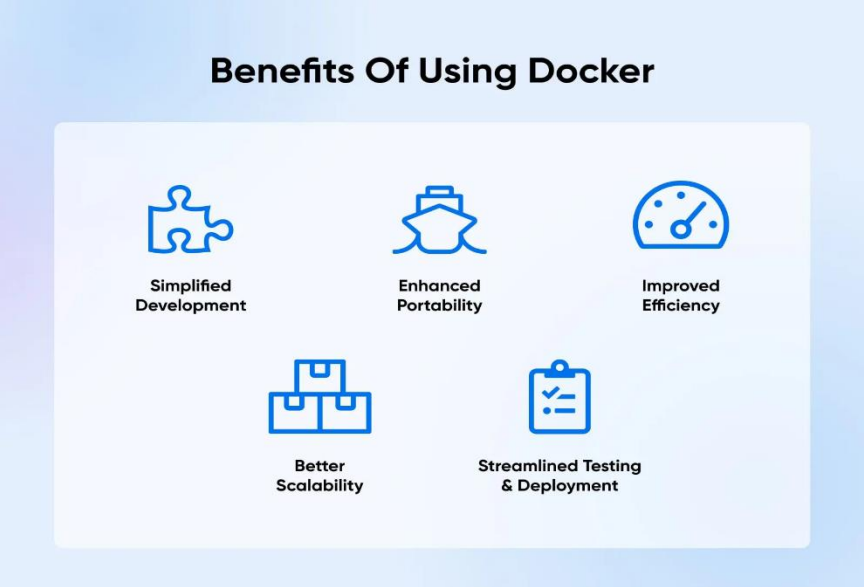
поддръжка. Софтуерът, който хоства контейнерите, се нарича **Docker Engine**. Проектът е започнат през 2013 г. и се разработва от Docker, Inc. След своето появяване, Docker значително започва да заменя виртуалните машини и бързо навлиза в софтуерната разработка, благодарение на своите предимства в ефективността и производителността.

- **Какво прави Docker уникален?**

Docker предлага стандартизирана и лесно достъпна среда за контейнеризация, която не само улеснява разработчиците, но и ускорява и автоматизира работата на DevOps екипите. След като вече описахме контейнерите по-горе, ето кои са ключовите предимства на Docker:

- **Преносимост**: Docker контейнерите могат лесно да се преместят между различни системи и среди, което решава проблемите със съвместимостта и гарантира, че приложението ще работи навсякъде по същия начин.
- **Лекота и бързина**: В сравнение с виртуалните машини, Docker контейнерите не включват отделна операционна система, което ги прави по-леки и значително по-бързи при стартиране.
- **Скалируемост и автоматизация**: Docker безпроблемно се интегрира с оркестрационни инструменти като Kubernetes, което улеснява управлението на големи клъстери от контейнери и автоматизира процесите по скалиране и разгръщане на приложения.
- **Консистентност на средата**: Docker осигурява, че приложението ще работи по един и същ начин във всички среди (разработка, тестови и продукционни среди), като елиминира несъответствията между различните конфигурации на операционните системи и гарантира стабилност при прехвърляне между тях.
- **Изолация**: Всеки контейнер е изолиран, което предотвратява конфликти между различни приложения или версии на същите зависимости. Това прави Docker идеален за микросървисни архитектури, където компонентите на приложението трябва да бъдат независими и отделени.
- **Ефективно управление на ресурси**: Docker контейнерите използват споделено ядро на операционната система, което елиминира нуждата от отделни операционни системи за всяко приложение, както при виртуалните машини. Това води до по-ниски разходи за ресурси и по-висока ефективност.
- **Лесно разгръщане и миграция**: Docker изображенията могат лесно да се качват в Docker Hub или други регистри, което прави миграцията и разгръщането на приложения бързо и лесно, особено при работа с облачни платформи или различни инфраструктури.
- **Поддръжка за CI/CD (непрекъсната интеграция и доставка)**: Docker се интегрира със процесите на автоматизирано тестване и внедряване, като осигурява идентични среди за тестове и продукция, което улеснява автоматизацията на целия жизнен цикъл на софтуера.

- **Управление на изображения**: Docker позволява ефективно управление на различни версии на изображения, което дава възможност на екипите да работят със специфични версии на приложенията и лесно да се върнат към предишни състояния при необходимост.



2. Основни принципи на работа: как Docker "пакетира" приложенията в контейнери

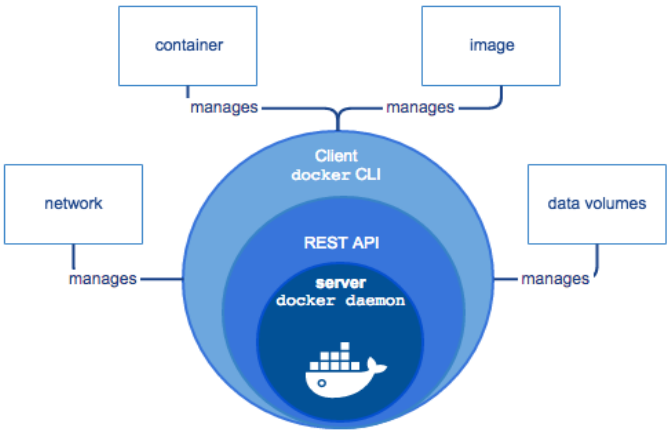
Docker използва различни технологии и концепции за създаване на изолирани среди за приложенията. Основните компоненти и процеси на Docker включват:

- **Docker изображения (Images)**: Docker изображенията са шаблони, които съдържат всички файлове, зависимости и конфигурации, необходими за стартиране на приложение. Те представляват основата, върху която се изграждат контейнерите. Изображенията са неизменни, което осигурява консистентност между различни среди и позволява лесно разпространение и повторно използване на същото изображение.
- **Контейнери (Containers)**: Контейнерите са работещите инстанции на Docker изображенията. Когато стартирате Docker изображение, се създава контейнер, който изпълнява приложението в изолирана среда. Контейнерите могат да бъдат управлявани независимо един от друг, а също и лесно пренасяни и мащабирани.
- **Dockerfile**: Dockerfile е текстов файл с инструкции, които дефинират как трябва да се изгради дадено Docker изображение. В него се описват стъпките за инсталиране на зависимости, копиране на файлове и настройка на конфигурации, което позволява създаването на персонализирани изображения, съобразени със специфичните нужди на дадено приложение.
- **Docker Hub и други регистри**: Docker Hub е публичен регистър, където се съхраняват и споделят Docker изображения. Това позволява лесен достъп до

голям брой готови изображения, включително официални изображения за популярни софтуерни решения.

- **Изолираност и мрежова свързаност**: Docker използва функции на ядрото, за да осигури изолация на контейнерите. Това означава, че всеки контейнер има собствена мрежова и файлова среда, което предотвратява конфликти и повишава сигурността на приложенията.
- **Управление и автоматизация**: Docker също така предоставя инструменти за управление и автоматизация на контейнерите, като например Docker Compose за дефиниране и изпълнение на многоконтейнерни приложения. Чрез Docker Compose разработчиците могат да зададат зависимости между контейнерите, като опростят и автоматизират работния процес.

Docker Engine: Обяснение на сървърната част и как работи тя с Docker Daemon



Docker Engine е основният компонент на Docker платформата, който отговаря за изграждането, изпълнението и управлението на контейнерите. Той включва както клиентска, така и сървърна част, като сървърната част е именно **Docker Daemon**.

Docker Daemon е основният процес, който работи във фонов режим на хост машината и управлява всички операции, свързани с контейнерите. Това включва задачи като изграждане на Docker изображения, стартиране, спиране и управление на контейнерите, както и взаимодействие с Docker клиентите и регистрите (например Docker Hub).

Основни функции на Docker Daemon:

- Управление на контейнери
- Изграждане на изображения
- Взаимодействие с Docker Hub
- Управление на мрежовата конфигурация

- Поддръжка на данни

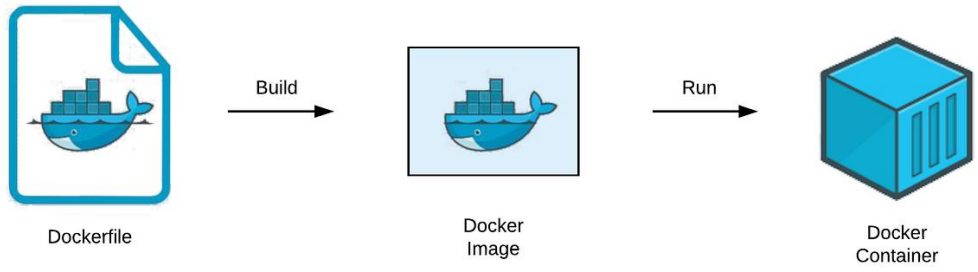
Как работи Docker Daemon с Docker Client?

Docker Client е интерфейсът, чрез който потребителят взаимодейства с Docker Daemon. Това е командният ред (CLI) или графичен интерфейс (например Docker Desktop), който изпраща команди към Docker Daemon за изпълнение на различни задачи, като например стартиране на контейнер, изграждане на изображение и други. Docker клиентът (CLI) изпраща заявки към Docker Daemon, който след това ги изпълнява. Процесът на комуникация между тях е следният:

1. **Изпращане на команди:** Потребителят използва Docker клиент (например, изпълнява команда docker run или docker build).
2. **Предаване на команди на Daemon:** Docker клиентът изпраща тези команди към Docker Daemon чрез HTTP API.
3. **Изпълнение от Daemon:** Docker Daemon получава командите и ги изпълнява, като създава или управлява контейнери, изгражда изображения, управлява мрежи и т.н.
4. **Резултати:** След изпълнението на командата Docker Daemon връща резултата обратно на Docker клиента (например информация за стартирането на контейнера, резултатите от изграждането на изображения или грешки, ако има такива).

Връзка с Docker API

Docker Daemon предоставя HTTP API, чрез което Docker клиентът може да комуникира с него. Това API позволява и автоматизация чрез различни инструменти и скриптове. API-то позволява на потребителите да взаимодействат с Docker Daemon чрез програмен код, което е основа за интеграция на Docker в различни DevOps процеси.



Преглед на базови Docker команди

`docker run ... -` стартира контейнер, базиран на дадено изображение и показва кратко съобщение, което потвърждава успешното стартиране на контейнера

`docker build ... -` използва се за изграждане на Docker изображение от Dockerfile. Това е процесът, чрез който се създава собствено изображение на база на зададените инструкции в Dockerfile

`docker pull ... -` служи за изтегляне на изображение от Docker registry, обикновено Docker Hub, но може да бъде и от вътрешни регистри

`docker ps -` показва всички текущо изпълняващи се контейнери

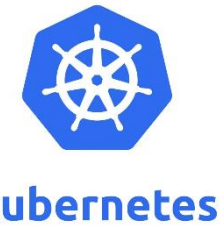
`docker stop [container_name] и docker start [container_name]-` спира контейнер / стартира спрян контейнер

`docker exec ... -` използва се за изпълнение на команди в работещ контейнер

`docker rm [container_name] и docker rmi [container_name]-` изтрива контейнер / изтрива Docker изображение

Официална документация на Docker команди и инструкции: Docker commands documentation

3. Kubernetes технология



След като вече разгледахме в детайли технологията Docker, сега ще обърнем внимание на друга много популярна технология в сферата на DevOps, облачната инфраструктура и софтуерната разработка, а това е именно **Kubernetes.**

- ### Дефиниция на Kubernetes като оркестрационна платформа за контейнери

Kubernetes е отворена платформа за оркестрация на контейнери, която автоматизира разгръщането, управлението, скалирането и функционирането на контейнеризирани приложения в клъстери от машини. Основната цел на Kubernetes е да улесни управлението на контейнери в големи мащаби, като осигури надеждност, преносимост и гъвкавост на приложенията в различни среди. С бързото развитие на контейнерните технологии, Kubernetes се превръща в **стандарт за оркестрация на контейнери** и основен елемент в облачните среди. Той се занимава с няколко важни аспекта на контейнеризацията:

- **Оркестрация на контейнерите**: Автоматичното управление и координация на контейнерите в клъстери, за да работят правилно и да отговарят на заявките.
- **Скалиране**: Автоматично увеличаване или намаляване на броя на контейнерите в зависимост от натоварването.
- **Самовъзстановяване**: Kubernetes рестартира контейнери при сривове и премества приложения на здрави възли, ако възникнат проблеми.
- **Равномерно разпределение**: Поддържа оптималното разпределение на ресурсите между приложенията.

• История на Kubernetes

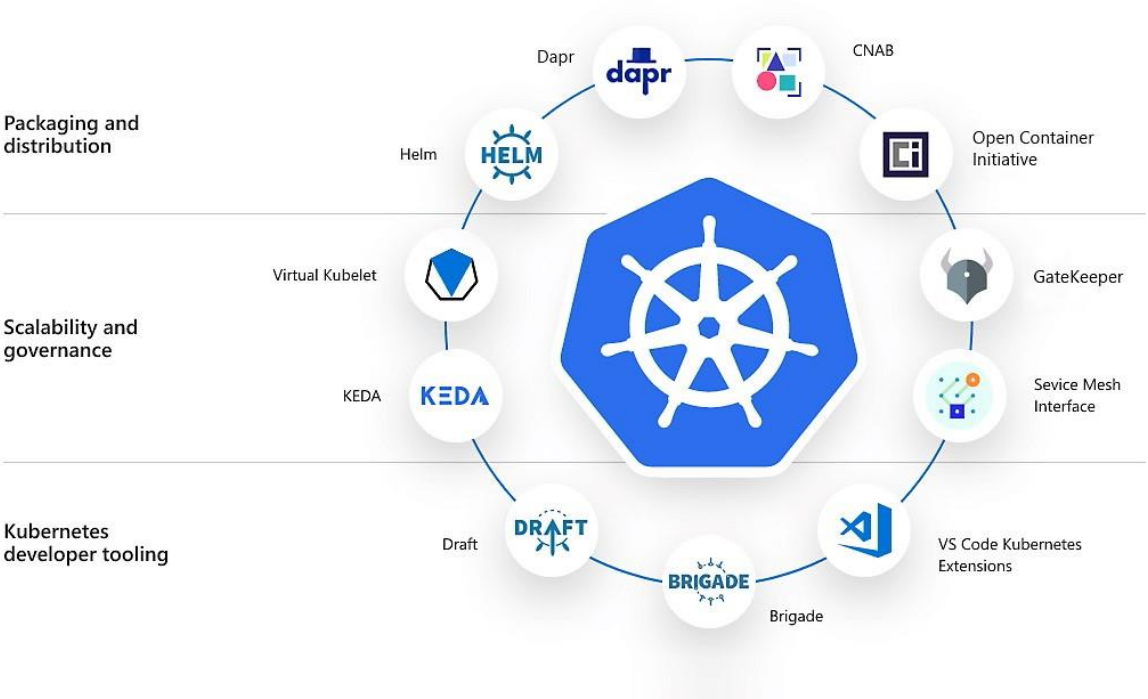
Kubernetes е пуснат през 2014г. (една година след Docker) и бързо става водеща платформа за оркестрация на контейнери. Неговите корени водят до вътрешна система на **Google**, наречена **Borg**, която Google използва за управление на милиони контейнери и приложения. С цел да направи тази технология достъпна за индустрията, Google решава да разработи нов, по-лесен за използване и преносим инструмент, като така създава Kubernetes като проект с отворен код (open-source). Проектът е разработен от **Cloud Native Computing Foundation** (CNCF), която подкрепя отворените облачни технологии и оркестрационни проекти. Kubernetes става основен проект на CNCF и започва да събира подкрепа от водещи компании в технологичния сектор, което допринася за широкото му приемане и внедряване. Написан е на езикът Go / Golang.

• Ролята на Google в създаването на Kubernetes и как той се е превърнал в стандарт за индустрията

Google играе ключова роля в създаването и разпространението на Kubernetes. Технологията, разработена в Google, за първи път предоставя на индустрията решение за оркестрация на контейнери, което е базирано на реални потребности от скалиране и ефективност. С пускането на Kubernetes като **проект с отворен код** Google цели да привлече общността и да създаде основа за иновации в управлението на контейнери. От момента на стартиране Kubernetes започва да се разпространява бързо в индустрията и скоро става стандарт за оркестрация на контейнери поради следните причини:

- **Отворен код и широката поддръжка от общността**: Като проект с отворен код Kubernetes е разработван и усъвършенстван с помощта на голям брой участници от различни компании и общности.
- **Поддръжка от водещи облачни доставчици**: Amazon, Microsoft, IBM, VMware и други бързо възприемат Kubernetes и започват да предлагат поддръжка за него в своите платформи, което го прави лесно интегрируем в различни облачни среди.

- **Гъвкавост и мащабируемост**: Kubernetes може да бъде внедрен както на локални машини, така и в облачни платформи, като предоставя много висока гъвкавост и надеждност за работни среди с високо натоварване.



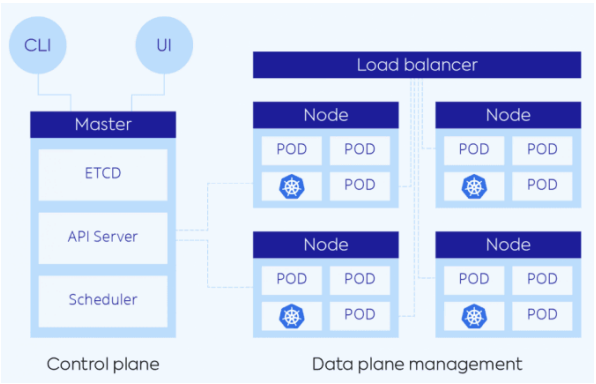
Компоненти и архитектура на Kubernetes

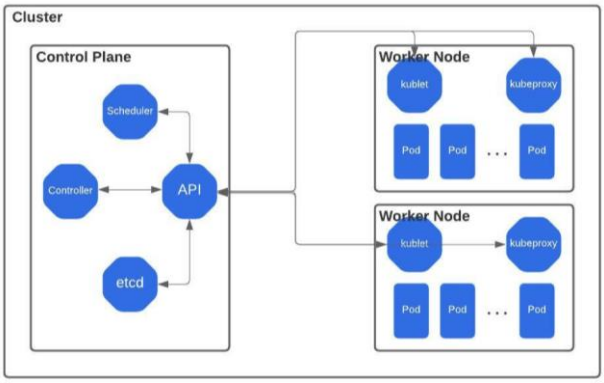
Kubernetes е сложна система, съставена от различни компоненти, които работят заедно, за да осигурят ефективна оркестрация на контейнери. Те са разделени на основни елементи, свързани с архитектурата и работата на системата.

- ## Основни компоненти на Kubernetes: Pods, Nodes, Clusters и Control Plane

1. **Pod**: Това е най-малката и основна единица в Kubernetes, която съдържа един или повече свързани контейнера, които споделят мрежа и място за съхранение. Контейнерите в Pod работят заедно и се разглеждат като едно логическо приложение. Ако даден контейнер спре да работи, Kubernetes може автоматично да стартира нов Pod със същите настройки.
2. **Node**: Представлява физически или виртуален сървър, който изпълнява Pods. Той съдържа Kubernetes компоненти като Kubelet, който комуникира с Control Plane, и Kube-proxy, който осигурява мрежовата комуникация. Нодовете са изпълнителните единици в Kubernetes клъстера и могат да се добавят и премахват според нуждите.
3. **Cluster**: Клъстерът обединява множество нодове и Control Plane, като образува цялостната инфраструктура за оркестрация на контейнери. Kubernetes клъстерът е съвкупност от нодове, които работят заедно и споделят ресурси. Той представлява средата, в която се управляват контейнерите.

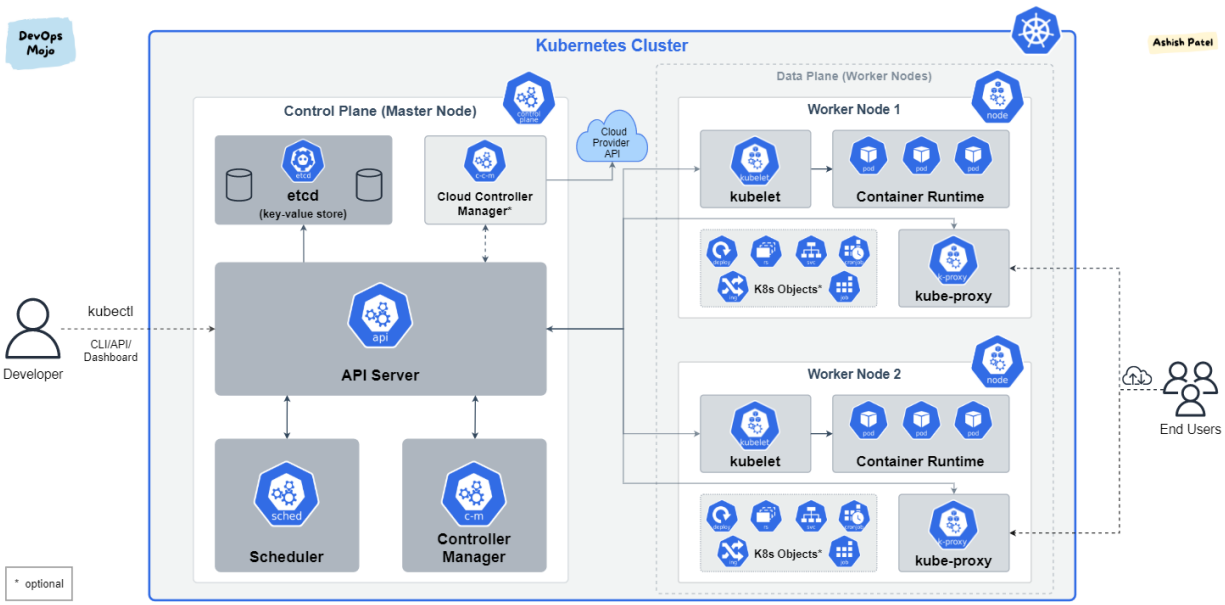
4. **Control Plane**: Control Plane е основната контролна част на Kubernetes, която управлява и координира всички нодове в клъстера. Control Plane комуникира с всички нодове и събира данни за тяхното състояние, като същевременно осигурява комуникацията с потребителите. Той съдържа няколко основни компонента, които се грижат за разпределението на ресурсите, планирането на задачите и скалирането.





- ## Ролята на компонентите в Control Plane: API Server, Scheduler, Controller Manager и etcd

1. **API Server:** API сървърът е интерфейсът за комуникация между потребителите, Control Plane и останалите компоненти на Kubernetes. Той приема заявки за създаване, актуализиране и изтриване на ресурси в клъстера и връща отговори за състоянието на тези ресурси. API Server е отговорен за осъществяването на всякакви взаимодействия със системата и предоставя RESTful интерфейс за управление на клъстера.
2. **Scheduler:** Това е компонентът, който решава къде в клъстера да бъдат разпределени нови Pods въз основа на нуждите на приложението и наличните ресурси. Той взема предвид редица фактори, включително натоварването на нодовете, местоположението на данните, и политики за разпределение, за да осигури равномерно и ефективно разпределение на приложенията.
3. **Controller Manager:** Обединява множество контролери, всеки от които наблюдава и управлява определени ресурси в Kubernetes. Контролерите в този компонент работят непрекъснато, за да поддържат желания статус на ресурсите в клъстера, като следят състоянието на нодовете и Pods, управляват репликациите и изпълняват автоматични корекции при възникване на проблеми.
4. **etcd:** Система за съхранение на ключ-стойност с висока достъпност, която пази състоянието на всички конфигурации и ресурси в клъстера. Тя осигурява централизирана база данни за Kubernetes и записва всички промени в клъстера, като поддържа консистентността на данните и улеснява координацията между компонентите. etcd гарантира, че данните за ресурсите са достъпни и съгласувани дори в случай на отказ на част от системата.



Основни концепции в Kubernetes

1. Deployments: Управление на множество реплики на Pods

Deployments са ключов Kubernetes обект, който улеснява управлението на мащабируеми и постоянно достъпни приложения. Те позволяват създаването и поддържането на множество реплики на Pod-овете, за да се гарантира устойчивостта на приложението дори при грешки или натоварване. Чрез Deployment конфигурация можем да посочим броя на репликите на Pod-овете, които Kubernetes трябва да поддържа, и системата автоматично ще осигури, че този брой остава постоянен.

2. Services и Ingress: Работа с мрежовия трафик и свързаността на Pod-овете

Kubernetes осигурява няколко мрежови обекта за улесняване на комуникацията вътре в клъстера и със света извън него.

- Services: Kubernetes Service обектите осигуряват постоянна мрежова връзка между Pod-овете и потребителите, или между самите Pod-ове. Когато се създаде Service, Kubernetes му назначава виртуален IP адрес, така че дори ако конкретни Pod-ове са прекратени и заменени с нови, връзката остава постоянна. Основните видове Services включват:
    - ClusterIP: Позволява вътрешна комуникация в клъстера.
    - NodePort: Отваря порт на всеки нод в клъстера за външен достъп до Service.

    - LoadBalancer: Предоставя външен IP адрес чрез облачен балансировчик на натоварването, полезен за разгръщане в облачни среди.
- Ingress: Ingress обектът осигурява правила за маршрутизиране на външния HTTP и HTTPS трафик към определени Services в клъстера. Ingress позволява детайлно управление на мрежовите пътища и правила за достъп, като насочва трафика към различни Services въз основа на конкретни домейни, пътища и HTTP методи. Това значително улеснява управлението на публично достъпни приложения и дава възможност за лесно създаване на правила за балансировка и защита.

3. Namespaces и тяхната роля за изолация в Kubernetes

Namespaces предоставят начин за логическа изолация на ресурси в един и същ Kubernetes клъстер. Те се използват основно за организиране и разделяне на ресурси, което е особено полезно за големи организации и клъстери с множество екипи и проекти. С Namespaces могат да се създадат независими пространства, в които различни приложения или среди да работят изолирано едно от друго. Namespaces позволяват също и по-ефективно управление на ресурси, като например квоти за CPU, памет и ограничаване на броя на обектите.

Namespaces се използват основно за:

- Разделяне на среди за разработка, тестване и продукция в един и същ клъстер.
- Изолиране на различни проекти и приложения на отделни екипи.
- Подобрена сигурност, като предоставя възможност за контрол над достъпа и ограничаване на ресурсите между отделните Namespaces.

Ползи на Kubernetes

Ще разгледаме 3-те основни ползи на тази технология, а именно: мащабируемост, гъвкавост и автоматизация

- **Мащабируемост**

Едно от най-големите предимства на Kubernetes е неговата способност за автоматично мащабиране на приложенията в отговор на натоварването. Kubernetes може автоматично да увеличава или намалява броя на репликите на дадено приложение (чрез Horizontal Pod Autoscaling), така че системата да издържи на увеличен трафик или да намали разхода на ресурси при по-малко натоварване. Тази гъвкавост позволява на екипите да реагират бързо на промени в търсенето, без да се налага ръчно добавяне на нови ресурси.

- **Гъвкавост**

Kubernetes осигурява възможност за работа с различни платформи и среди, включително локални сървъри, облачни услуги и хибридни конфигурации. Благодарение на своята модулност, Kubernetes може да бъде адаптиран към различни инфраструктурни нужди и работни натоварвания, което позволява интеграция с различни системи и среди. В допълнение, Kubernetes поддържа различни видове приложения – от микросървисни архитектури до монолитни системи.

- **Автоматизация**

Kubernetes предлага автоматизация на редица процеси, като внедряване на приложения, мониторинг, откриване на услуги и самовъзстановяване. Kubernetes Deployment обектите позволяват автоматично внедряване на нови версии на приложенията чрез rolling updates, което позволява безопасни промени без спиране на услугите. Kubernetes автоматизира също така и процесите по възстановяване на повредени Pod-ове, балансиране на натоварването и дори управление на мрежовия трафик. Всичко това води до по-малка нужда от ръчна намеса, което повишава ефективността и намалява риска от грешки.

Ограничения на Kubernetes

Основните ограничения на технологията са сложността на конфигурация и нуждата от дълбоки познания на ниво администрация. Kubernetes е сложна платформа, която изисква задълбочено разбиране и планиране за успешно внедряване и поддръжка. Конфигурирането на елементите също е времеемко и изисква специфични познания, както и техническа експертиза. За да се осигури надеждна поддръжка и оптимално функциониране на системата, е необходимо администраторите да познават добре Kubernetes и да имат опит в управлението на клъстери, мрежи, съхранение и сигурност.

4. Връзка между технологиите Docker и Kubernetes

Docker и Kubernetes са взаимно допълващи се технологии, които работят съвместно за опростяване на разгръщането, управлението и мащабирането на приложения в съвременни DevOps среди. Docker предоставя стандартизирана платформа за

контейнеризация, докато Kubernetes осигурява средства за оркестрация и управление на тези контейнери в клъстер от машини. В DevOps контекста комбинацията от двете технологии прави процесите по-бързи, по-надеждни и лесни за автоматизация и именно поради това комбинацията от двете технологии е много често срещана.

В следващата част ще разгледаме как точно двете технологии си взаимодействат:

1. Как Kubernetes използва Docker контейнери за разгръщане на приложения

Kubernetes се основава на Docker контейнери за изолиране и стартиране на приложения, като предоставя възможност за автоматично управление на тези контейнери в голям мащаб. Kubernetes използва Docker контейнери за разгръщане на приложения по следния начин:

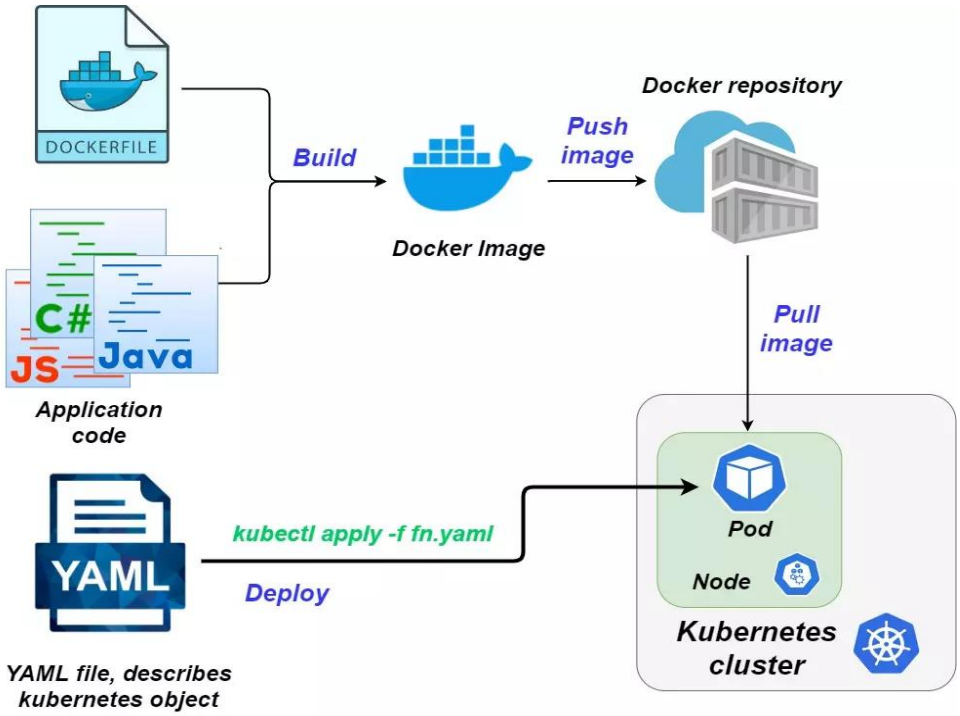
- **Изолирано изпълнение**: Docker пакетира приложенията и техните зависимости в изолирани контейнери, които работят независимо от хост системата. Kubernetes се възползва от тази изолация, като стартира всеки контейнер в отделни Pod-ове, които работят в рамките на един клъстер. Това осигурява надеждност и контрол над всяко приложение.
- **Управление на конфигурацията и мрежовата свързаност**: Docker предоставя стандартен начин за управление на контейнерите, като мрежова конфигурация и достъп до хранилища. Kubernetes добавя слой за оркестрация, който автоматично настройва мрежовата свързаност и хранилищата, необходими на Docker контейнерите, като същевременно се грижи за изолирането на различни подове.
- **Мащабиране на приложения**: Kubernetes използва Docker контейнери за поддръжка на динамично мащабиране на приложения. Kubernetes може автоматично да добавя или премахва копия (реплики) на подове с Docker контейнери въз основа на натоварването. Това осигурява ефективност и позволява на приложенията да отговорят на промените в трафика и натоварването.

2. Връзката между Docker и Kubernetes в една реална DevOps среда

Docker и Kubernetes често се използват заедно като част от DevOps практиките поради начина, по който техните функции се допълват. Тази комбинация от технологии предоставя мощна инфраструктура за автоматизация и гъвкаво управление на жизнения цикъл на приложенията.

- **Контейнеризация с Docker за разработка и тестове**: В DevOps средите разработчиците често използват Docker за създаване на контейнеризирани версии на приложенията си. Това позволява на екипите да създават стандартизирани контейнери, които могат да се използват във всички етапи на жизнения цикъл – от локалната разработка, през тестове, до продукция.
- **Оркестрация с Kubernetes за автоматизация и мащабируемост**: Kubernetes предоставя необходимите инструменти за автоматизиране на управлението и разгръщането на Docker контейнерите. В DevOps среда, това позволява на екипите да внедряват непрекъснати актуализации на приложенията (CI/CD) чрез автоматизиране на процесите по разгръщане и скалиране.

- **Управление на средата и устойчивост**: Kubernetes поддържа изолирането на приложения и управлението на мрежовия трафик, което е изключително важно за DevOps практиките. Docker осигурява стабилна и последователна среда за стартиране на приложенията, а Kubernetes автоматизира възстановяването при грешки, управление на ресурси и баланс на натоварването. Това осигурява стабилност на приложенията и надеждност на цялата инфраструктура.
- **Подобрена ефективност и скорост на разработка**: Съчетавайки Docker и Kubernetes, DevOps екипите могат да работят с една и съща среда от разработка до продукция. Това улеснява сътрудничеството между разработчици, тестъри и администратори, като елиминира проблемите със съвместимостта и ускорява процеса на разработка и внедряване.



Работен процес: Как Docker контейнери се управляват от Kubernetes

За успешното разгръщане на контейнеризирани приложения в Kubernetes, работният процес започва със създаването на Docker контейнер и преминава през няколко етапа, за да се постигне автоматизирано управление и мащабиране на приложението. В следващите няколко стъпки ще демонстрираме процеса на работа на двете технологии:

1. Създаване на Docker контейнер

Процесът започва с разработката на приложението и неговата контейнеризация с помощта на Docker. За да се създаде контейнер е необходимо следното:

- **Пише се Dockerfile** в който се описва как да бъде пакетирано приложението – включително неговите зависимости, конфигурации и инструкции за стартиране.

- **Създаване се Docker изображение**, което представлява статична, неизменна версия на приложението, което съдържа всичко необходимо за неговото стартиране. Изображението може да бъде качено в Docker Hub или в частен регистър, от който Kubernetes може да го изтегли по-късно.

2. Подготовка за разгръщане в Kubernetes

След като Docker контейнерът е създаден и качен в регистър, следващият етап е интеграцията му в Kubernetes чрез конфигурация на различни Kubernetes обекти:

- **Създаване на Kubernetes манифестни файлове**: За да разгръщат и управляват Docker контейнерите в Kubernetes, администраторите създават манифестни файлове на YAML или JSON, които описват как ще се разпределят и управляват контейнерите в Kubernetes.
- **Определяне на Deployment конфигурация**: В Kubernetes Deployment манифестът позволява на екипа да посочи реплики на приложението, стратегия за обновяване и други параметри, които контролират как Kubernetes ще управлява контейнера след разгръщане.

3. Разгръщане и управление на Docker контейнери в Kubernetes

След като манифестните файлове са готови, разгръщането и управлението на контейнерите в Kubernetes се извършва чрез различни автоматизирани стъпки:

- **Разгръщане в Kubernetes клъстера**: Манифестните файлове се подават на Kubernetes с командата kubectl apply, което стартира процеса по разгръщане на Docker контейнерите в клъстера. Kubernetes създава подове, съдържащи Docker контейнерите, като използва Docker изображенията от предварително зададения регистър. Подовете се разпределят върху наличните worker nodes в клъстера, в зависимост от ресурсите и политиките за разпределение.
- **Контрол върху контейнерите и наблюдение**: Kubernetes автоматично следи състоянието на подовете и контейнерите. Ако даден контейнер спре или излезе от строя, Kubernetes автоматично го рестартира, за да осигури непрекъснатост на работата. Това се осъществява чрез контролната равнина на Kubernetes, която следи и координира ресурсите на клъстера, използвайки компоненти като Scheduler и Controller Manager.

4. Мащабиране и обновяване

След разгръщането, Kubernetes може да мащабира Docker контейнерите автоматично, като същевременно поддържа висока достъпност и последователност:

- **Автоматично мащабиране**: Kubernetes може да променя броя на активните подове според натоварването чрез Horizontal Pod Autoscaler, като гарантира оптимално използване на ресурсите и добра производителност на приложението.
- **Rolling updates и възстановяване**: Kubernetes Deployment манифестът позволява безопасно обновяване на Docker контейнерите с нови версии чрез rolling updates. При тази стратегия Kubernetes постепенно актуализира всеки под, така че приложението да не прекъсва работата си. Ако новата версия не работи правилно,

Kubernetes автоматично връща промените и възстановява предишната стабилна версия на контейнера.

Разлики между Docker и Kubernetes

В този раздел ще направим сравнение между Docker и Kubernetes като обобщим тяхната основна цел и ще обърнем внимание на често срещаните погрешни представи за двете технологии и начина по който си взаимодействат.

1. **Основна цел**:
    - **Docker** е платформа за контейнеризация, която създава изолирани контейнери за приложения, включващи всички необходими зависимости.
    - **Kubernetes** е оркестрационна платформа за управление на контейнери в голям мащаб, като координира тяхното разгръщане, мащабиране и автоматизация в клъстер.
2. **Функционалност**:
    - **Docker** създава и управлява контейнери, но самостоятелно не може да мащабира приложения и не предлага оркестрация на контейнерите.
    - **Kubernetes** управлява контейнерите на множество сървъри, като гарантира автоматично мащабиране, баланс на натоварването и възстановяване при срив.
3. **Подход за управление**:
    - **Docker** се използва за локална разработка и тестване на приложения в контейнери.
    - **Kubernetes** е подходящ за сложни среди с много контейнери, които трябва да се разпределят и управляват в клъстери.

4. Обхват на управление

- **Docker**: Docker самостоятелно оперира на ниво *индивидуален сървър*. Той не е създаден да координира работата на множество хостове и не може ефективно да управлява клъстер от сървъри.
- **Kubernetes**: Kubernetes управлява *клъстер от машини* (независимо дали са физически или виртуални), като разпределя контейнерите между тях според наличните ресурси и нуждите на приложенията. Това позволява на приложенията да бъдат разпределени по множество сървъри, за да се осигури висока наличност и мащабируемост.

5. Модел на съхранение на данни

- **Docker**: Docker поддържа различни методи за съхранение на данни, като „bind mounts“ и „volumes,“ но Docker самостоятелно не предлага разпределено съхранение на данни. Възстановяването на данни между различни контейнери или клъстери изисква външно решение.
- **Kubernetes**: Kubernetes има по-сложна система за управление на обемите (Persistent Volumes), като позволява съхранение и прехвърляне на данни между подове и поддържа различни доставчици на облачно съхранение.

Погрешни представи за Docker и Kubernetes

1. **„Kubernetes е алтернатива на Docker“**: Погрешно е да се смята, че Kubernetes заменя Docker. Docker се фокусира върху контейнеризацията, а Kubernetes върху оркестрацията на контейнерите. Двете технологии обикновено работят заедно.
2. **„Docker и Kubernetes се използват само в облака“**: Макар че двете технологии са често асоциирани с облачни среди, те могат да се използват и в локални инфраструктури.
3. **„Kubernetes е лесен за настройка“**: Kubernetes предлага множество функции за мащабиране и автоматизация, но изисква високо ниво на техническа експертиза за настройка и управление, което го прави по-сложен за внедряване и поддръжка в сравнение с Docker.

5. Реален пример за съвместна работа на Docker и Kubernetes

В този раздел ще представя примерен базов проект, който демонстрира как Docker и Kubernetes работят заедно за създаване, разгръщане и управление на приложение. Следните стъпки ще илюстрират процеса по изграждане на Docker контейнер за уеб приложение и неговото разполагане в Kubernetes клъстер. Този пример цели да покаже как Docker и Kubernetes си взаимодействат, за да постигнат разгръщане и мащабируемост на уеб приложение.

- ### Следвам следните стъпки:

1. **Изграждане на Docker изображение за уеб приложение** – създавам просто уеб приложение на Python (Flask framework)
2. **Публикуване на изображението в Docker Hub** – изображението го качвам в Docker Hub, което позволява достъпност за Kubernetes
3. **Създаване на Kubernetes Deployment и Service** - конфигурацията на Deployment и Service ще използва Docker изображението, за да разположи приложението в клъстера.
4. **Тестване в локален Kubernetes клъстер (напр. Minikube)**

Описание на проекта: Разгръщане на Flask уеб приложение с Docker и Kubernetes

Обща цел на примера: Проектът включва създаване на уеб приложение с Python и Flask, което ще се постави в Docker контейнер и след това ще се разположи в Kubernetes клъстер.

- **Необходими стъпки за изпълнение на проекта:**

1. **Разработка на Flask приложението:**

В директорията на проекта създавам файла app.py със следния код:

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello from Flask in Docker and Kubernetes!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

2. Създаване на Dockerfile

Създавам Dockerfile със следното съдържание, за да поставя приложението в Docker контейнер:

```
FROM python:3.9-slim

WORKDIR /app
COPY app.py /app

RUN pip install Flask

EXPOSE 5000

CMD ["python", "app.py"]
```

3. Изграждане на Docker изображение и локално тестване

Изпълнявам следните команди за изграждане и тестване на Docker изображението:

```
docker build -t my-flask-app .

docker run -p 5000:5000 my-flask-app
```

4. Публикуване на Docker изображението в Docker Hub

```
docker login

docker tag my-flask-app valeryraikov/my-flask-app

docker push valeryraikov/my-flask-app
```

5. Разгръщане в Kubernetes

Подготовка на Deployment и Service конфигурации: Създавам файл deployment.yaml със следната конфигурация:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask-container
        image: valeryraikov/my-flask-app
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  selector:
    app: flask-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

6. Стартиране на приложението в Kubernetes клъстера:

```
kubectl apply -f deployment.yaml
```

7. Достъп и скалиране на приложението в Kubernetes

След успешното разгръщане, проверяваме статуса на приложението:

```
kubectl get deployments

kubectl get pods

kubectl get services
```

8. Бърза проверка в браузъра:

Hello from Flask in Docker and Kubernetes!

6. Заключение

- **Обобщение на ползите от използването на Docker и Kubernetes заедно**

Комбинацията на Docker и Kubernetes предоставя изключителни предимства за съвременната софтуерна разработка, особено при внедряването и управлението на приложения в мащабни производствени среди. Docker улеснява създаването на самостоятелни контейнери, които осигуряват среда за работа, независима от специфичната инфраструктура, на която ще се разгръщат. Това дава възможност за бързо разработване, тестване и конфигуриране на приложения, които са лесно преносими и изолирани едни от други.

Когато Docker се комбинира с Kubernetes, системните администратори и разработчиците получават пълен контрол върху разпределението, балансирането на натоварването и автоматичното възстановяване на контейнерите. Kubernetes улеснява създаването на клъстери от контейнери и разширява възможностите за автоматизация на инфраструктурата, като осигурява плавен процес на мащабиране и балансиране на натоварването. Това е особено ценно за големи системи и приложения, които се нуждаят от непрекъснатост и висока надеждност.

Съвместната работа на двете технологии позволява на екипите да изградят напълно автоматизиран процес на CI/CD (непрекъсната интеграция и доставка), който значително намалява времето за внедряване на нови версии и поддържа еднаква среда за всички фази на разработката.

Използвана литература:

- https://en.wikipedia.org/wiki/Docker_(software)
- https://en.wikipedia.org/wiki/Kubernetes
- https://docs.docker.com/
- https://kubernetes.io/docs/home/
- https://www.geeksforgeeks.org/introduction-to-docker/
- https://www.trianz.com/insights/containerization-vs-virtualization
- https://www.aquasec.com/cloud-native-academy/docker-container/containerization-vs-virtualization/
- https://k21academy.com/docker-kubernetes/kubernetes-vs-docker/