MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

VIRTUALIZAREA APLICAȚIILOR CU DOCKER ȘI KUBERNETES

LUCRARE DE LICENŢĂ

Absolvent: **Dan Andrei MERCEAN**

Coordonator ştiinţific: **Asis. ing. Cosmina IVAN**

2018

Cuprins

Capitolul 1. Introducere – Contextul proiectului

În ultimii ani, serviciile web au devenit din ce în ce mai populare. Internetul este în cea mai mare amploare și toate aplicațiile încep să fie rulate pe el. Aplicațiile de tip desktop sunt mutate pe web, accesate din browser, ajungându-se să ai nevoie doar de acesta pentru a face orice.

Dacă serviciile furnizate de o corporație nu mai sunt disponibile, un mare impact apare asupra acesteia, date se pierd și clienții devin nemulțumiți. Datorită naturii serviciilor web, frecvența de utilizare a aplicațiilor diferă pe perioade de timp. O metodă de a rezolva această variație este planificarea capacității, adică determinarea cantității de resurse necesare pentru a îndeplini calitatea serviciilor. Dacă se ia în considerare numărul maxim de resurse necesare, unele resurse nu o să fie folosite în momentul în care traficul pe aplicație este mic. Un număr mare de resurse neutilizate duce la un consum mare de energie și la deteriorarea componentelor hardware fără niciun folos. Producția și instalarea noilor componente hardware este costisitoare pentru o companie sau organizație când vine vorba de climat și resurse financiare. De asemenea, cu cât numărul de calculatoare și servicii este mai mare, timpul necesar administratorilor de sistem pentru gestiunea lor este mai mare.

Pentru a rezolva problema resurselor neutilizate se folosește alocarea dinamică. Astfel, numărul necesar de resurse va fi alocat pentru fiecare perioadă de timp. În acest mod, aplicația este capabilă să își păstreze calitatea și în momentele de maximă cerere, dar va utiliza și mai puține resurse în momentele cu cerere mai redusă. Tehnologia care permite acest lucru este virtualizarea. Împărțind resursele disponibile în mașini virtuale, administratorii sunt capabili să izoleze aplicațiile importante în infrastructură și să utilizeze resursele într-un mod mai eficient decât în cazul în care aplicațiile erau separate pe mașini fizice diferite.

De la apariția extensiei arhitecturii hardware a procesoarelor X86 care permite virtualizarea hardware, virtualizarea a avut o creștere enormă pe calculatoarele cu o arhitectură X86, în particular, dezvoltarea Kernel-based Virtual Machine (KVM) pentru sistemele de operare Linux, sau creșterea interesului pentru cloud computing. Principalele beneficii ale utilizării virtualizării sunt consolidarea serverelor, izolarea și o gestiune mai simplă. Aceasta permite utilizatorilor să aibă sisteme de operare concurente pe un singur calculator, pe care le poate gestiona de la un singur terminal.

Bifurcarea mașinilor virtuale este procesul de clonare a unei mașini virtuale în mai multe replici, care pot să ruleze pe host-uri diferite. Acest lucru este benefic pentru serviciile web cu trafic fluctual pentru că permite utilizarea mai multor replici la nevoie. Pot să fie create noi mașini virtuale și prin utilizarea unui șablon preconfigurat. Există șabloane bazate pe imagini sau șabloane bazate pe text. Un șablon imagine se bazează pe o mașină virtuală, pe baza căreia sunt inițializate noi mașini virtuale. Un șablon text are o sintaxă specifică care conține caracteristicile unei mașini virtuale.

Docker, un nou tip de software bazat pe containere Linux, este cea mai mare concurență a mașinilor virtuale. O aplicație și dependențele sale pot să fie împachetate într-un container, care poate să fie distribuit și rulat după cerințele de trafic. În timp ce containerele Linux sunt virtualizate, ele nu virtualizează și resursele hardware. Totodată, containerele oferă aceeași izolare a aplicațiilor ca mașinile virtuale și pot să fie oprite și pornite ca aplicații. Resursele folosite de un container pot să fie schimbate și configurate

în timp ce acesta încă rulează. Acest lucru duce la mari beneficii în momentul în care este vorba de scalare.

Kubernetes este un sistem *open-source* pentru automatizarea implementării, scalării și gestionării aplicațiilor containerizate. Acesta grupează containerele care alcătuiesc o aplicație în unități logice pentru o gestionare și descoperire ușoară. Kubernetes se bazează pe o experiență de 15 ani din rularea volumelor de lucru în producție la Google, combinat cu idei și practici de cea mai bună calitate din comunitate. Proiectat pe aceleași principii care permit celor de la Google să ruleze miliarde de containere pe săptămână, Kubernetes poate scala fără a crește echipa de operatori. Indiferent dacă se testează la nivel local sau dacă se rulează o întreprindere globală, flexibilitatea Kubernetes livrează aplicațiile în mod consecvent și ușor, indiferent cât de complexe sunt necesitățile acesteia. Kubernetes este *open source,* oferind libertatea de a profita de infrastructura cloud locală, hibridă sau publică, făcând posibilă mutarea fără efort a volumului de lucru oriunde este necesar.

Capitolul 2. Obiectivele Proiectului

Obiectivul principal al acestei lucrări este descoperirea tehnicilor existente de virtualizare care folosesc containerele și înțelegerea acestora. Principalele conceptele care se vor urmări sunt: *chroot* (comanda care schimbă directorul de lucru implicit), *jail* (mecanism de izolare), *namespaces* (spațiile de nume) și *cgroups* (grupurile de control). Se vor prezenta principalele uneltele care folosesc virtualizarea cu containere și se vor analiza comparativ cu metodele de virtualizare care folosesc *hypervisor*.

Se va aprofunda cel mai cunoscut și folosit program de containerizare, Docker, analizând arhitectura acestuia, librăria proprie pentru containere (*libcontainer*), sistemul de fișiere, probleme de securitate și proiecte în care este folosit. De asemenea, se urmărește înțelegerea utilitarului care gestionează grupurile de containere, Kubernetes, împreună cu conceptele care stau la baza funcționalității sale, cum ar fi crearea componentelor folosind fișiere YAML, actualizarea aplicațiilor containerizate sau scalarea lor.

Obiectivul studiului este pregătirea contextului de execuție a unei aplicații prin implementarea tehnicilor de virtualizare analizate. Se urmărește virtualizarea unei aplicații folosind serviciile de containerizare furnizate de Google Kubernetes Engine. Se va utiliza o aplicație scrisă în Angular TypeScript, care folosește un server Apache Tomcat și o bază de date MySQL. Se va utiliza o stocare persistentă pentru baza de date MySQL, Deployment-ul și Service-ul MySQL, cât și al serverului Apache Tomcat și al aplicației Angular Web. Această virtualizare se va realiza și folosind Minikube pentru a demonstra că o aplicație containerizată cu Docker și Kubernetes este portabilă.

Se dorește testarea virtualizării oferită de Minikube și Google Kubernetes Engine prin măsurarea timpului necesar unui Deployment și serviciu Tomcat până în momentul în care facilitățile oferite de ele erau disponibile.

Capitolul 3. Studiu Bibliografic

3.1. Virtualizarea bazată pe containere

Furnizarea unui mediu izolat într-un sistem de operare gazdă este cunoscut ca virtualizarea la nivelul sistemului de operare și un exemplu de astfel de mediu poate fi definit ca un container: un mediu de execuție de sine stătător, care împarte nucleul sistemului gazdă și este izolat de restul containerelor din sistem.

3.1.1. Chroot și jail

Scopul creării comenzii *chroot* a fost izolarea proceselor de fișierele sistemului gazdă. Fiind prescurtarea de la *change root*, această comandă specifică un nou director sursă, altul decât cel implicit (/). Niciun proces nu poate accesa fișiere din exteriorul noii surse, în timp ce programele din exterior pot să vadă în interiorul ei. Este crucial ca niciun proces din interior să nu poată obține privilegiile sursei, deoarece acestea pot să permită ieșirea din respectivul director. În zilele noastre, *chroot* este folosit pentru a furniza medii izolate de testare a aplicațiilor necunoscute sau instabile.

Un mecanism avansat construit peste *chroot* este *jail*. Acesta adaugă izolare a listelor de procese, a seturilor de utilizatori și networking. Așadar, *jail* poate defini un nou utilizator sursă, care are control deplin în interior, dar care nu poate ajunge în exterior. Limitările sunt incapacitatea de a monta sau demonta sisteme de fișiere și modificarea configurării rețelei. Elementele *jail* pot să fie pornite, oprite sau restartate și, cu ajutorul utilitarului *ezjails,* chiar arhivate pentru folosirea lor mai târziu.

3.1.2. Namespace

Namespace (spațiul de nume) este caracteristica cheie a nucleului Linux pentru suportarea virtualizării *lightweight*. Scopul acestora fiind împachetarea unei resurse globale a sistemului într-o abstracție care face procesele din interior să creadă că au propria instanță izolată a resursei globale. În nucleul Linux se pot identifica mai multe tipuri de spații de nume:

- *Mount namespace* a fost primul spațiu implementat și face vizibile toate montările și demontările sistemelor de fișiere din spațiul global. Orice astfel de operație din interiorul unui spațiu nu este vizibil nici unui alt spațiu. Propagarea acestora s-ar putea realiza cu o relație master-slave.
- *Unix timestamp sharing* (UTS) permite izolarea identificatorilor *gethostname()* și *getdomainname(),* împreună cu membrii corespondenți din *uname()*. Orice modificare făcută cu vreunul dintre aceștia este vizibilă doar în interiorul spațiului celui care le apelează.
- *Interprocess communication* (IPC) furnizează izolarea memoriei partajate, a semafoarelor și a diferitelor sisteme de fișiere pentru coada de mesaje POSIX (Portable Operating System Interface).
- *Process ID* (PID) izolează lista identificatorilor proceselor. Permite proceselor din diferite spații să dețină același PID și combinarea acestor spații. Principiul de bază este ca un proces dintr-un spațiu să cunoască doar procesele din spațiul său sau din spațiile inferioare lui. Primul proces din orice spațiu are PID 1 și, când un

proces este terminat, toate procesele sale orfane devin copiii procesului cu PID 1. Acest proces nu poate fi terminat cu niciun semnal.

- *Network namespaces* permite fiecărui spațiu să dețină propria stivă de rețea, cum ar fi propriile adrese IP, numere de port, tabele de rutare sau dispozitive de rețea. Când este creat un nou spațiu de rețea, acesta conține doar dispozitivul de *loopback* (*lo*), dar dispozitivele de rețea pot fi mutate printre spațiile de rețea. Regula este ca, orice dispozitiv în afară de *lo*, să aparțină unui singur spațiu de rețea. Combinarea spațiilor de rețea este de asemenea posibilă.
- *User namespaces* permite proceselor să dețină utilizatori și identificatori de grupuri diferiți în interiorul spațiului. Fișierele */$proc_id/uid_map* și */proc/$proc_id/gid_map* conțin maparea actuală a intervalului de identificatori a utilizatorilor sau a grupurilor de la spațiul copil la cel părinte. Beneficiul este că, chiar și un utilizator neprivilegiat poate să pornească un proces privilegiat, în interiorul unui spațiu. Este suportată și combinarea acestora.

Implementarea spațiilor a adăugat două apeluri sistem noi, *setns()* pentru a adera la un spațiu deja existent și *unshare(),* care permite proceselor apelante continuarea într-un spațiu nou creat. Apelul *sistem clone()* a adăugat 6 noi *flag*-uri, unul pentru fiecare spațiu, pentru a permite proceselor copil rezultate începerea într-un spațiu nou creat.

3.1.3. Control groups

Folosind spațiile de nume, procesele sunt izolate între ele, dar nu suficient față de ceea ce oferă un sistem de virtualizare standard. Pe lângă creare și rulare, este necesară gestiunea mașinilor virtuale. *Control groups* (grupurile de control) este un nucleu care permite administatorilor restricționarea sau limitarea utilizării resurselor sistem pentru grupuri de procese. Grupurile pot avea o structură de tip arbore, unde rădăcinile sunt grupurile implicite ale fiecărui sistem.

Subsistemele sunt controlorii care se referă la tipurile de resurse individuale. Configurațiile fiecărui grup subsistem sunt stocate în sistemul de fișiere (de obicei */cgroups/subsystem/),* unde se află setările grupurilor implicite. Setările grupurilor copil sunt stocate în același loc, recursiv, în directoare adiționale. Subsistemele cele mai folosite sunt:

- *blkio* – controlează accesul la dispozitivele *block io* și poate fi folosit pentru a limita viteza de citire sau scriere, pentru a seta prioritatea unui grup la citire sau scriere, global sau pentru fiecare dispozitiv;
- *cpu* – atribuie puterea CPU pe total, setând prioritatea CPU sau timpul absolut în care un proces poate rula;
- *cpusets* – specifică CPU-uri individuale pe care un grup le poate folosi;
- *devices* – permite crearea unei liste cu drepturi de acces la dispozitive individuale;
- *memory* – impune limite asupra memoriei folosite de procesele din grup.

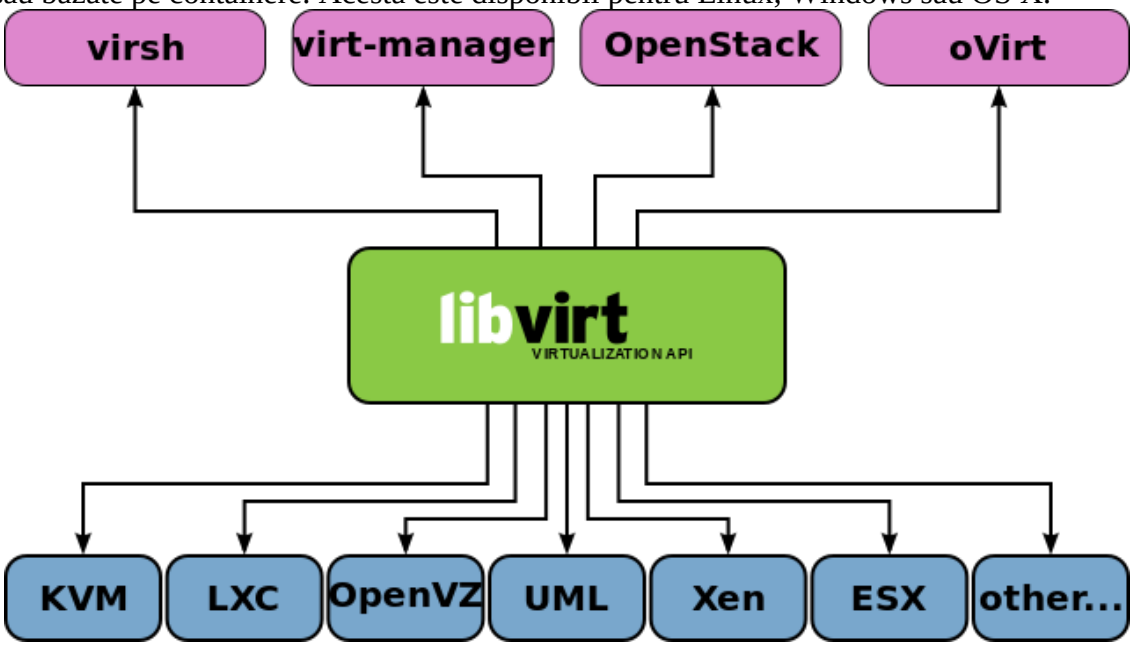
3.1.4. Instrumente bazate pe virtualizare

Containerele Linux (LXC) combină caracteristicile oferite de spațiile de nume și grupurile de control pentru a crea un mediu complet izolat, care poate fi gestionat cu ușurință. Operațiile permise sunt crearea, pornirea, oprirea, listarea și ștergerea acestora. Crearea containerelor este posibilă folosind șabloane, care setează sistemul de fișiere și

configurația inițială a containerului. În plus, este posibilă înghețarea și dezghețarea containerelor pentru a suspenda execuția unui container care se poate relua mai târziu. Crearea de puncte de verificare permite restaurarea stării containerelui la starea sa din momentul efectuării acestei verificări. O altă funcționalitate care poate fi folosită este clonarea containerelor.

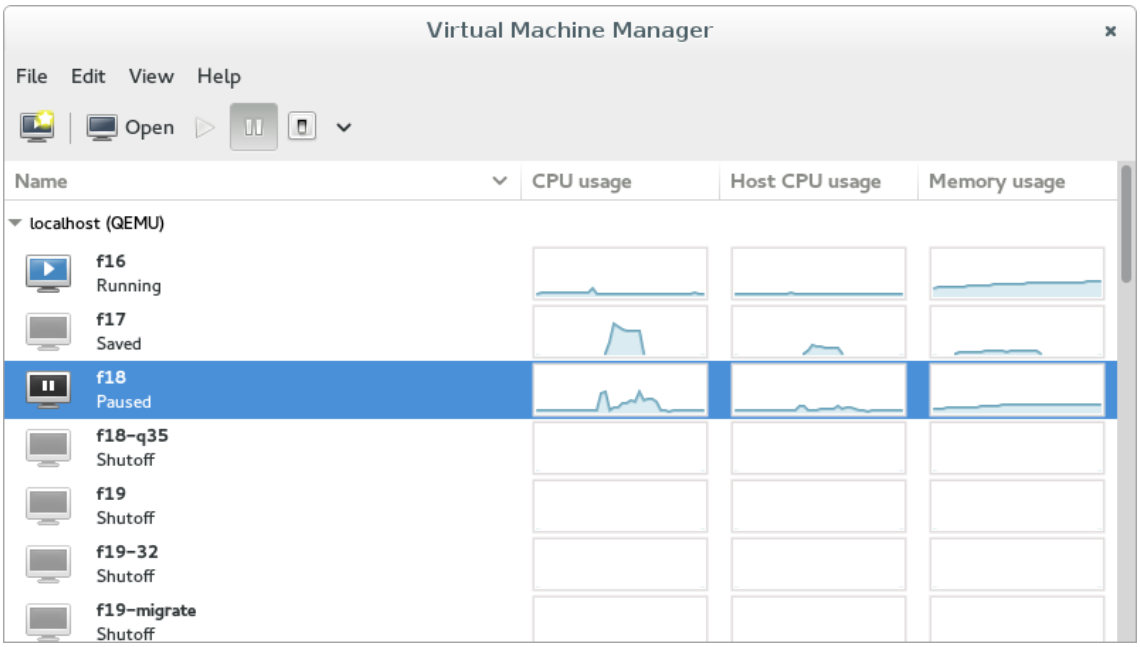
OpenVZ, o altă tehnologie de virtualizare bazată pe containere, este un strămoș al proiectului LXC, deoarece mult cod din LXC derivă din OpenVZ sau membrii echipei OpenVZ au contribuit la dezvoltarea acestuia. Diferența dintre cele două tehnologii este faptul că proiectul OpenVZ folosește propriul nucleu, derivat din nucleul Linux 2.6. Mulți administratori de sistem preferă OpenVZ, deoarece acesta a fost folosit cu succes în producție pentru o perioadă mare de timp. LXC nu suportă anumite funcționalități, cum ar fi migrarea dinamică (suspendarea unui container pe un host și rezumarea lui pe un alt host), din acestă cauză, OpenVZ este mai des folosit pentru a furniza servere private virtuale.

Gestiunea mașinilor virtuale poate deveni o sarcină dificilă, deoarece există multe instrumente de virtualizare, astfel încât o companie este posibil să dețină mai multe mașini virtuale sau containere create cu diferiți furnizori. Pentru a depăși această problemă, *libvirt* (ilustrat în Figura 3.1) furnizează un API (Application Programming Interface) unificat, suportând o gamă largă de instrumente de virtualizare, fie ele standard sau bazate pe containere. Acesta este disponibil pentru Linux, Windows sau OS X.



Figură 3.1 *Hypervisor*-ii care sunt suportați de libvirt și soluțiile de gestiune care il suportă pe acesta

Virt-manager (Virtual Machine Manager), prezentat în Figura 3.2, este un instrument de gestiune a mașinilor virtuale construit de Red Hat peste *libvirt*, care vine cu o interfață grafică bogată. Acesta este disponibil doar pentru Linux.



Figură 3.2 Interfața grafică a Virtual machine manager

3.1.5. Comparație cu mașinile virtuale

Mașinile virtuale crează noi instanțe ale sistemelor de operare pentru fiecare mașină pe care o rulează. Acest lucru aduce beneficii, precum rularea unui sistem de operare oaspete diferit de sistemul de operare gazdă. Un dezavantaj ar fi faptul că ocupă mult spațiu pe disc sau că mentenanța lor este dificilă. Containerele necesită doar aplicația și dependențele sale, în timp ce nucleul este împărțit între ele. Pentru că sistemul de operare rulează deja, pornirea unui container este mult mai rapidă decat pornirea unei mașini virtuale. Un nucleu împărțit poate să nu fie întotdeuna un beneficiu, un exemplu fiind incapacitatea de a rula aplicații Windows în containere Linux.

3.2. Docker

Ideea din spatele Docker a fost crearea unui „buton” care să permită construirea oricărei aplicații și lansarea ei pe orice server, oriunde. Acesta este o platformă open-source care permite lansarea aplicațiilor în interiorul containerelor. Acest proiect a atras repede atenția lumii IT, făcând companii ca Google, Amazon, Microsoft sau Red Hat să îl suporte și să contribuie la dezvoltarea lui.

Docker este mai mult decât o librărie pentru virtualizare, acesta abstractizează diferențele dintre sistemele de operare creând un mediu standard pentru dezvoltarea aplicațiilor. Un programator poate să creeze o aplicație standardizată, care devine portabilă și poate rula oriunde există mecanismul Docker instalat. Acest lucru salvează mult timp, deoarece autorul nu mai este nevoit să suporte diferite platforme sau sisteme de operare. Administratorii sistemului petrec mai putin timp configurând aplicația, deoarece aceasta vine însoțită cu toate dependențele sale. Rularea fiecărei aplicații în containerul său rezolvă probleme cum ar fi nevoia unei dependențe între două aplicații cu versiuni diferite.

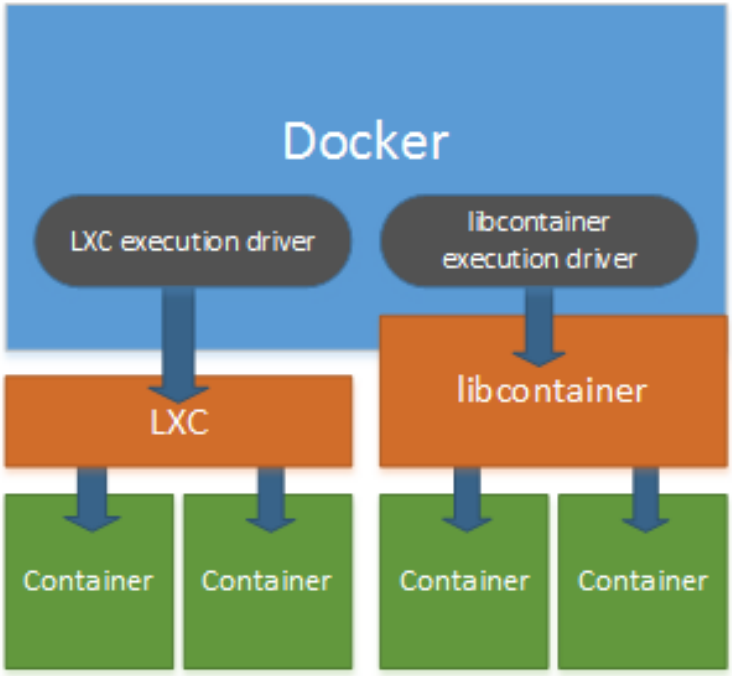
Docker este o tehnologie nouă, care are limitările sale. Acesta suportă doar aplicații care pot rula pe Linux și poate rula nativ doar pe Linux. Rularea pe Windows sau OS X necesită un software adițional și o mașină virtuală.

3.2.1. Docker Daemon

Intern, Docker folosește un model client-server, unde serverul este un *daemon* care rulează pe un sistem total diferit față de client. Acesta poate fi pornit folosind comanda *docker* cu *flag*-ul *-d* sau cu *systemctl start docker*. Pentru a fi rulat sunt necesare privilegii *root*, chiar daca se depune efort pentru a permite folosirea lui și de către utilizatorii regulari.

3.2.2. Libcontainer

În versiunea 0.9 a fost introdus conceptul de *execution driver*, fiind suportate două astfel de motoare după cum este ilustrat și în Figura 3.3: motorul LXC, care folosește *liblxc* și motorul nativ Docker, care folosește propria librărie, *libcontainer*. Scris complet în Go, acesta gestionează containere folosind capacitățile nucleului, cum ar fi spațiile de nume sau grupurile de control. *Libcontainer* dorește să fie portat în alte limbaje și să suporte diferite sisteme de operare, astfel eliminând necesitatea de software adițional precum Boot2docker atunci când Docker este rulat pe Windows.

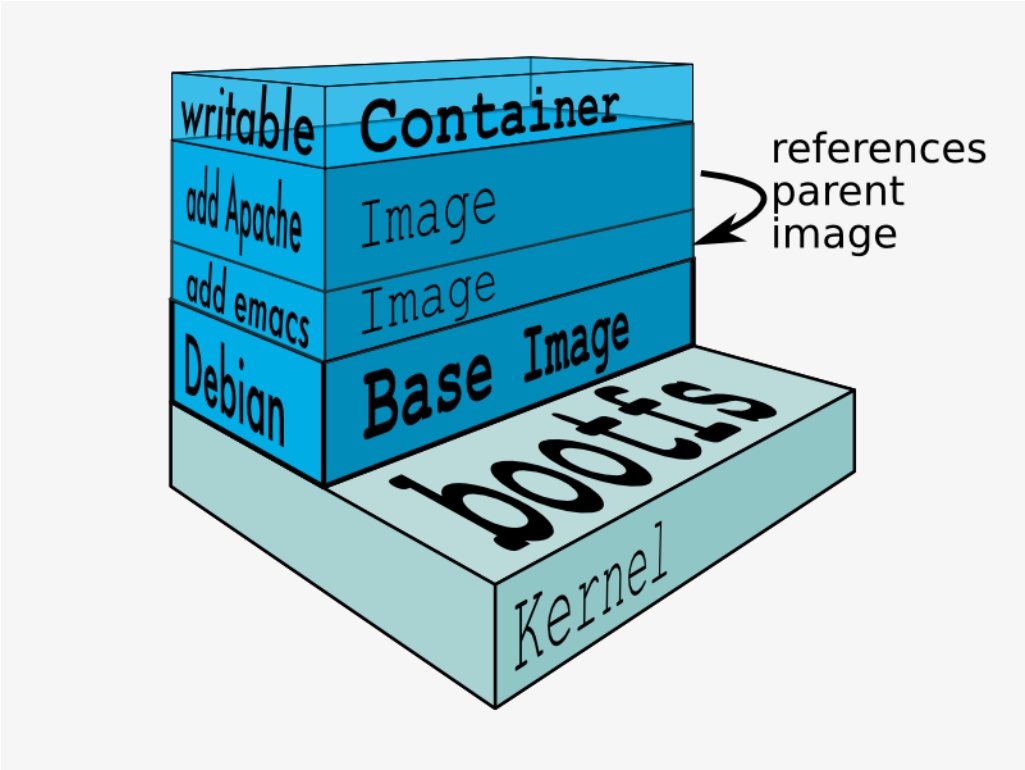


Figură 3.3 Docker Execution Driver

3.2.3. Sistem de fișiere pe nivele

Când un container este lansat, Docker folosește un mecanism numit *union mount* (sistemele de fișiere nu sunt montate în locuri diferite ci unul peste altul, Figura 3.4), deci conținutul unui director poate fi compus din fișierele directoarelor altor sisteme de fișiere.

În Docker, aplicațiile specifică o imagine sursă. De exemplu, o aplicație web poate să depindă de un server web specific, care depinde doar de un sistem de operare. Asta înseamnă că orice imagine adaugă un nou nivel, *read only*, deasupra nivelului său sursă. O dată ce aplicația este pornită ca un container, este adăugat un nivel adițional peste restul nivelelor, nivel în care se poate scrie. Când un fișier dintr-un nivel *read only* trebuie să fie modificat, acesta se copiază în ultimul nivel și se modifică. Este important ca schimbările din acest nivel să persiste chiar și după oprirea unui container, acestea fiind necesare la următoarea rulare.



Figură 3.4 Sistemul de fișiere Docker

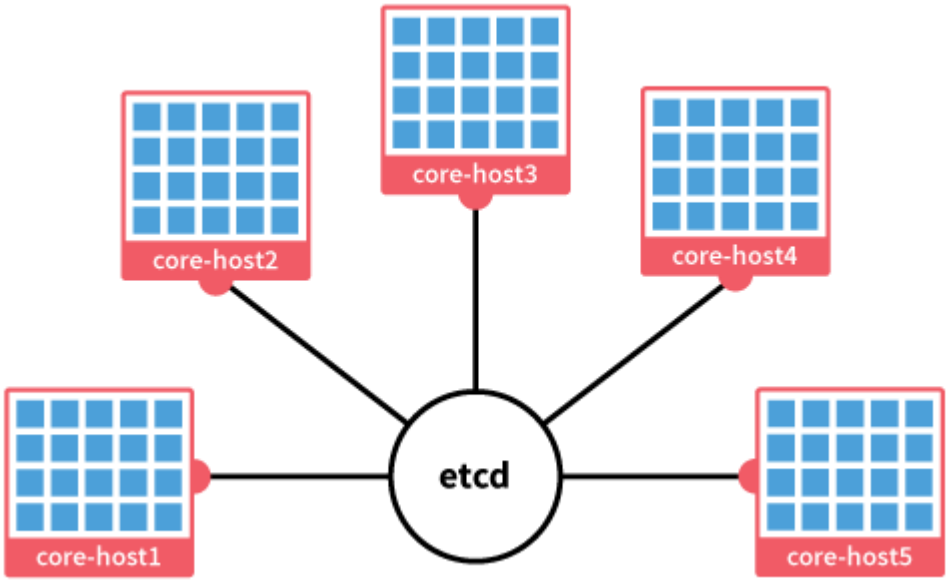
3.2.4. Securitate

Docker are nevoie de privilegii *root*, motiv pentru care, dacă acesta este compromis, *host*-ul este compromis și el. Din cauza acestei amenințări a securității, se lucrează la abilitatea acestuia de fi rulat de utilizatori regulari. Structura necesară este constituită din doi *daemons*, unul care să ruleze în spațiul utilizator și unul privilegiat, care să ruleze în spațiul nucleului.

În Decembrie 2014, Docker a promovat semnarea imaginilor, deoarece s-a dorit de mult o semnătură criptografică pentru a verifica imaginile înaintea rulării lor. S-a dovedit că Docker verifică doar existența unui manifest semnat, nu și valoarea acestuia. Acest lucru făcea posibilă rularea oricărei imagini care avea asociată un manifest, deoarece valoarea salvată în acesta nu era verificată. O altă problemă a fost faptul că, deși manifestul era greșit, imaginea era rulată oricum, fiind creată doar o avertizare. Ca urmare a numărului mare de vulnerabilități, echipa Docker a promis revizuirea securității.

3.2.5. Alte proiecte care folosesc containere

CoreOS este un sistem de operare mic, open-source, bazat pe sistemul de operare Chrome de la Google. Acesta este un sistem de operare centrat pe containere, toate aplicațiile fiind furnizate în containere. Ținta acestui sistem de operare este infrastructura *could*, având la bază două utilitare cheie: *etcd* și *fleet*. *Etcd* este folosit pentru gestiunea configurației împărțită de grupuri, furnizând un API pentru propagarea schimbărilor configrației peste tot grupul de instanțe *etcd*. *Fleet* este *daemon* pentru controlul sistemului la nivelul grupului. Acesta permite lansarea containerelor global sau pe o singură mașină. În figura de mai jos se poate observa un grup de host-uri CoreOS, creat de *fleet* și configurat de *etcd*.



Figură 3.5 Grup de host-uri CoreOS

Snappy Ubuntu Core este un alt sistem de operare care rulează aplicații containerizate. Acesta încearcă să furnizeze un sistem de operare minimal capabil să ruleze aplicații bazate pe containere. El aduce actualizări atomice sistemului, care se aplică aplicațiilor și lui însuși. Actualizările au forma unor tranzacții, deci este posibilă anularea lor. Intern, toate versiunile de bază ale pachetelor sunt salvate, actualizările aducând doar diferențele față de versiunea trecută. Ubuntu dezvoltă și LXD (Linux Contained Daemon), care dorește folosirea tuturor capacităților Docker pentru a furniza un API care să includă migrare în timp real și abilitatea de a rezuma un container dintr-un anumit punct. Acesta este un fel de extensie pentru LXC.

Echipa CoreOS a considerat că proiectul Docker a luat-o pe un drum greșit, acesta trebuia să fie un standard pentru containerizare, o componentă simplă, o unitate compusă, care să poată fi utilizată într-o varietate de sisteme. Membrii echipei au considerat că scopurile proiectului au devenit prea largi, vorbind mai degrabă despre o platformă, Docker Platform. Principalele lor griji erau proiectele Docker Machine, Docker Swarm și Docker Compose. Astfel, echipa a decis lasarea proiectului Rocket, executor de containere și competitor direct cu Docker. Acesta a fost redenumit *rkt* (*rock it*) și a propus

implementarea specificațiilor aplicațiilor containerizate folosite de ei pentru a face cele două proiecte interoperabile. Proiectul a continuat incluzând ambele executoare în sistem.

3.3. Kubernetes

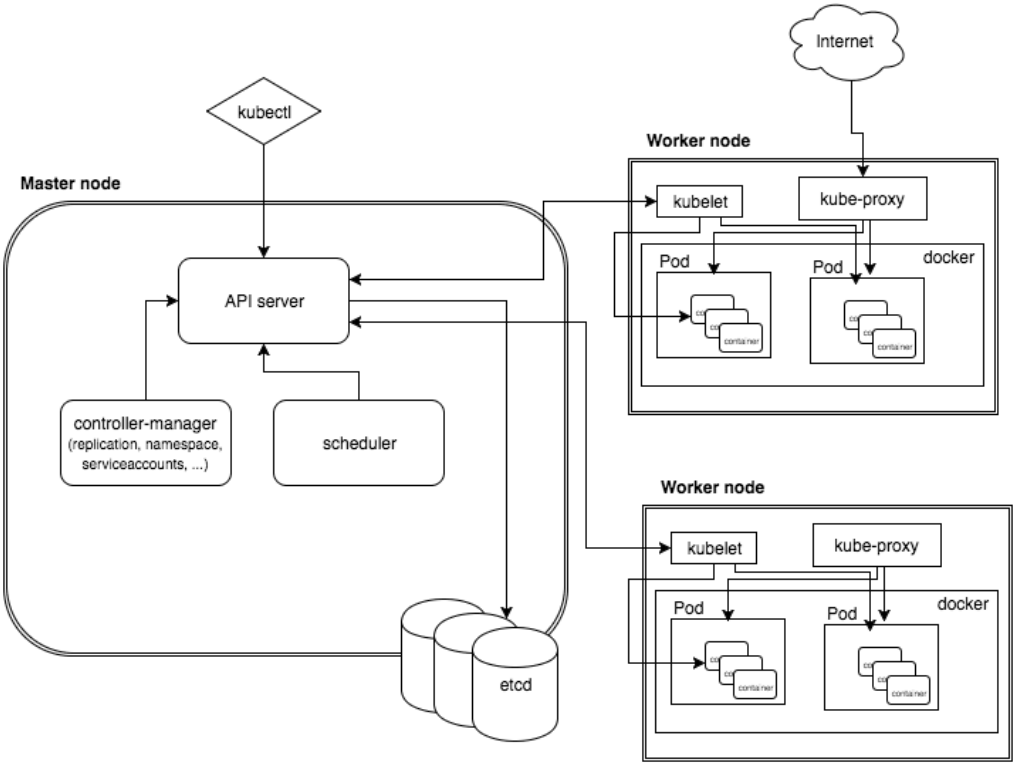
Kubernetes este un proiect open-source foarte mare, cu mult cod și multe funcționalități. Acesta a fost lansat de Google ca parte a unui efort de a împărți propria infrastructură și avantajele tehnologice cu întreaga comunitate. Aceștia lansează miliarde de containere pe saptămână în infractructura lor și folosesc tehnologia containerizată de peste un deceniu. La început, dezvoltatorii Google construiau un sistem numit Borg (numit Omega acum), pentru a planifica volumul lor mare de lucru peste *data center*-urile lor. De-a lungul anilor au învățat multe lecții și astfel au rescris unealta de gestiune pentru adoptarea pe scală largă, la nivel mondial. Rezultatul a fost proiectul open-source Kubernetes.

Principala responsabilitate a Kubernetes este orchestrarea containerelor. Acest lucru presupune ca toate containerele să fie programate să ruleze pe mașini fizice sau virtuale. Containerele trebuie să fie împachetate eficient după constrângerile mediului de lansare și configurația grupului. Adițional, Kubernetes trebuie să supravegheze toate containerele care rulează și să înlocuiască containerele finalizate, deteriorate sau care nu răspund.

Kubernetes plasează automat containerele pe baza cerințelor de resurse și a altor constrângeri, fără a sacrifica disponibilitatea. Se combină volumele de lucru critice și volumele de lucru cu cele mai bune eforturi pentru a ajuta la utilizarea și economisirea a cât de multe resurse. Acesta repornește containerele care au eșuat, înlocuiește și reprogramează containerele când nodurile mor și nu le face publice clienților până când nu sunt gata să servească solicitările acestora. Nu este necesară modificarea aplicației pentru a utiliza un mecanism de descoperire a serviciilor necunoscut. Kubernetes oferă containerelor propriile lor adrese IP și un singur nume DNS pentru un set de containere, echilibrând sarcina de lucru între ele. Este posibilă scalarea aplicației în sus și în jos printr-o comandă simplă sau automat bazându-se pe utilizarea procesorului. Kubernetes derulează progresiv modificările aplicației sau ale configurației acesteia pentru a se asigura că nu ucide toate instanțele în același timp. Dacă ceva nu merge bine, Kubernetes poate reveni la versiunea anterioară. Sistemul de gestiune permite crearea și actualizarea secretelor și a configurației aplicației fără a reface imaginea și fără a expune secretele în stiva de configurație. Sistemul de stocare este montat automat, de la un depozit local, un furnizor de cloud public sau un sistem de stocare în rețea.

3.3.1. Concepte Kubernetes

Chiar dacă Docker furnizează nivele de abstractizare și unelte pentru gestiunea containerelor, Kubernetes furnizează ajutor similar pentru orchestrarea grupurilor de containere și gestiunea stivelor de aplicații (ilustrate în Figura 3.6). Acesta oferă construcții pentru a face managementul la nivelul aplicației sau a serviciului. Automatizarea și uneltele sunt furnizate pentru a asigura disponibilitatea mare, stiva de aplicații și portabilitarea la nivel de serviciu. Kubernetes oferă și un control mai bun a utilizării resurselor peste infrastructură.



Figură 3.6 Arhitectura Kubernetes

Kubernetes furnizează nivele de abstractizare pentru orchestrarea gestiunii prin construcții cheie pentru combinarea containerelor, punctelor finale și a datelor în stive de aplicații și servicii. Kubernetes oferă și unelte pentru a gestiona când, unde și câte componente sunt în stivă. Sunt folosite conceptele de stare dorită și stare actuală pentru gestiunea colecțiilor și a volumelor de lucru. Toate componentele care formează Kubernetes lucrează constant la monitorizarea stării curente și la sincronizarea acesteia cu starea dorită. Uneori, sincronizarea nu este realizabilă, dar sistemul lucrează mereu la aceasta. Conceptele care stau la baza componentelor Kubernetes sunt:

a) *Cluster* - acesta este o colecție de resurse de depozitare și de rețea pe care le folosește Kubernetes pentru a rula volumele de lucru ale sistemului. Un sistem poate să conțină mai multe grupuri.

b) *Node* - Un nod este un singur host, fie el fizic sau virtual. Rolul lui este de a rula *Pod*-uri. Fiecare nod Kubernetes rulează câteva componente, cum ar fi *kubelet* și *kube proxy*. Nodurile sunt gestionate de Kubernetes master și sunt albinele muncitoare ale Kubernetes care fac toată munca grea. Acestea erau numite și minioni. *Kubelet* interacționează cu serverul API pentru a actualiza starea și pentru a începe volumele de lucru noi care au fost invocate de programator. *Kube proxy* asigură echilibrarea încărcării și direcționează traficul destinat anumitor servicii către suportul corespunzător din backend.

c) *Master* - acesta este creierul grupului, unde găsim serverul API, care menține servicii web *RESTful* pentru interogarea și definirea clusterului dorit și a stării de lucru. Controlul accesează numai *master*-ul pentru inițierea modificărilor, fără a fi posibilă accesarea directă a nodurilor. În plus, *master*-ul include planificatorul, care

lucrează impreună cu serverul API pentru a programa încărcările de lucru, sub formă de *Pod*-uri, pe nodurile minion. Aceste *Pod*-uri includ diferite containere care alcătuiesc stivele de aplicații. În mod implicit, planificatorul Kubernetes răspândește *Pod*-uri în întregul *Cluster* și folosește noduri diferite pentru replici de *Pod*. Kubernetes permite și specificarea resurselor necesare pentru fiecare container, astfel încât programarea poate fi modificată de acești factori suplimentari. Controlerul de replicare funcționează cu serverul API pentru a se asigura că numărul corect de replici de *Pod* rulează la un moment dat. Dacă controlerul de replicare definește trei replici și starea reală este de două copii, atunci planificatorul va fi invocat pentru a adăuga un al treilea *Pod* undeva în *Cluster*. Același lucru este valabil dacă există prea multe *Pod*-uri care rulează în *Cluster* la un moment dat. În cele din urmă, *etcd* rulează ca un magazin distribuit de configurare. Starea Kubernetes este stocată aici și *etcd* permite urmărirea valorilor. Aceasta este memoria partajată a creierului.

d) *Pod* - acesta este unitatea de lucru în Kubernetes. Fiecare *Pod* conține unul sau mai multe containere. *Pod*-urile sunt întotdeauna programate împreună (funcționează întotdeauna pe aceeași mașină). Toate containerele dintr-un *Pod* au aceeași adresă IP și un spațiu pentru porturi; pot comunica folosind localhost sau comunicarea standard între procese. În plus, toate containerele dintr-un *Pod* pot avea acces la spațiul de stocare local partajat pe nodul care găzduiește *Pod*-ul. Spațiul de stocare partajat va fi montat pe fiecare container. *Pod*-urile sunt o caracteristică importantă din Kubernetes, acestea furnizează o soluție excelentă pentru gestiunea grupurilor de containere care sunt înrudite, care depind unele de altele și trebuie să coopereze pe același host pentru a-și îndeplini scopul. Este important să ținem minte că *Pod*-urile sunt temporare, entități care pot fi aruncate sau înlocuite după buna voință. Orice stocare a unui *Pod* este distrusă o dată cu el. Fiecare *Pod* primește un identificator unic pentru a fi posibilă o diferențiere între ele.

e) *Label* - Etichetele (Labels) sunt perechi cheie-valoare care sunt utilizate pentru a grupa seturi de obiecte, foarte des *Pod*-uri. Acest lucru este important pentru alte concepte, cum ar fi controlerul de replicare, seturile de replici și serviciile care funcționează pe grupuri dinamice de obiecte și care trebuie să identifice membrii grupului. Există o relație NxN între obiecte și etichete. Fiecare obiect poate avea etichete multiple și fiecare etichetă poate fi aplicată la obiecte diferite. Există anumite restricții pentru etichete: fiecare etichetă de pe un obiect trebuie să aibă o cheie unică. Cheia de etichetare trebuie să respecte o sintaxă strictă și are două părți: prefix și nume. Prefixul este opțional, dacă există, atunci este separat de nume printr-un slash (/) și trebuie să fie un subdomeniu valid DNS. Prefixul trebuie să aibă cel mult 253 de caractere. Numele este obligatoriu și trebuie să aibă maxim 63 de caractere. Numele trebuie să înceapă și să se termine cu un cod alfanumeric (a-z, A-Z, 0-9) și conține numai caractere alfanumerice, puncte și liniuțe. Valorile respectă aceleași restricții ca și numele. Etichetele sunt dedicate pentru identificarea obiectelor și nu pentru atașarea de metadate arbitrare la obiecte.

f) *Annotation* - acestea permit asocierea metadatelelor arbitrare cu obiecte Kubernetes. Kubernetes doar stochează adnotările și face metadatele disponibile. Spre deosebire de etichete, nu au restricții stricte cu privire la caracterele permise și limitele de dimensiune. Acestea sunt necesare în sistemele complexe, motiv pentru care, Kubernetes le-a oferit deja implementate, pentru a nu fi nevoie de propriul depozit de metadate.

g) *Label selector* (selectorii de etichete) - sunt utilizați pentru a selecta obiecte pe baza etichetelor sale. Selectorii bazați pe egalitate specifică un nume de cheie și o valoare. Există doi operatori, = (sau ==) și ! =, pentru egalitate sau inegalitate bazată pe valoare, de exemplu: *rol = server de web*. Aceasta va selecta toate obiectele care au cheia și valoarea etichetei respective. Selectorii de etichete pot avea cerințe multiple separate printr-o virgulă, de exemplu: *rol = webserver, application! = foo*. Selectorii bazați pe seturi extind capacitățile și permit selectarea bazată pe multiple valori: *rol in* (webserver, backend).

h) *Replication controller și replica set* (controlorul de replicare și seturile de replici) - gestionează un grup de minioni identificați de un selector de etichete și asigură funcționarea unui anumit număr dintre aceștia. Principala diferență dintre ele este aceea că controlorii de replicare testează apartenența după nume și seturile de replici pot utiliza selecția bazată pe set. Seturile de replici sunt mai noi și desemnate ca controlori de replicare pentru generația următoare.

i) *Service* - acestea sunt folosite pentru a expune anumite funcționalități utilizatorilor sau altor servicii. De obicei, ele cuprind un grup de *Pod*-uri, identificate de o etichetă. Pot fi servicii care să ofere acces la resurse externe sau la *Pod*-uri controlate direct la nivelul IP virtual. Serviciile funcționează la nivelul 3 (TCP / UDP). Kubernetes 1.2 a adăugat obiectul Ingress, care oferă acces la obiecte HTTP. Serviciile sunt publicate sau descoperite prin intermediul a două mecanisme: DNS sau variabile de mediu. Serviciile pot fi echilibrate de Kubernetes, dar dezvoltatorii pot alege să gestioneze ei echilibrarea în cazul serviciilor care utilizează resurse externe sau a celor care necesită un tratament special.

j) *Volume* - depozitarea locală pe *Pod* este temporară și dispare o dată cu *Pod*-ul. Depozitarea lor nu este necesară, dacă se face doar schimb de date între containerele din nod, dar în alte cazuri este important ca datele să supraviețuiască *Pod*-ului sau este necesară transmiterea datelor între *Pod*-uri. Conceptul volumelor susține această nevoie. Există mai multe tipuri de volume, iar Kubernetes susține fiecare tip de volum existent. Tipul volumului *emptyDir* montează un volum pe fiecare container care este suportat în mod implicit de orice este disponibil pe maşina gazdă. Dacă este necesar, poate fi solicitat un mediu de memorie. Acest spațiu de stocare este șters când *Pod*-ul este terminat din orice motiv. Există mai multe tipuri de volume pentru anumite medii de tip *cloud*, diverse sisteme de fișiere în rețea și chiar depozite Git. Un tip de volum interesant este *persistentDiskClaim*, care abstractizează detaliile și utilizează stocarea implicită din mediu.

k) *StatefulSet* - uneori, Kubernetes trebuie să gestioneze depozite de date distribuite cum ar fi MySQL. Aceste grupuri de depozite păstrează datele distribuite pe noduri identificate în mod unic. Acest lucru nu poate fi făcut cu *Pod*-uri normale și servicii. *StatefulSet* asigură (similar unui controlor de replicare) ca un număr de poduri cu identități unice să ruleze. *Pod*-urile au următoarele proprietăți: un nume de gazdă stabil, disponibil în DNS; un index de ordine și stocare stabilă legată de numărul de ordine și gazdă. *StatefulSet* poate ajuta la descoperirea perechilor și la adăugarea sau înlăturarea *Pod*-urilor.

l) *Secret* - acestea sunt obiecte mici care conțin informații sensibile, cum ar fi credențialele și tokenurile. Ele sunt stocate ca plaintext în *etcd*, accesibile de către serverul Kubernetes API, și pot fi montate ca fișiere în *Pod*-uri (folosindu-se volume

secrete dedicate peste volumele obișnuite de date) care au nevoie de acces la ele. Același secret poate fi montat în *Pod*-uri multiple. O altă abordare este utilizarea de secrete ca variabile de mediu. Secretele dintr-un *Pod* sunt întotdeauna stocate în memorie pentru o securitate mai bună.

m) *Name* - fiecare obiect din Kubernetes este identificat printr-un UID și un nume. Numele este folosit pentru a se face referință la obiectul din apelurile API, trebuie să aibă o lungime de până la 253 de caractere și să utilizeze caractere alfanumerice mici, liniuțe și puncte. Dacă se șterge un obiect, se poate crea un alt obiect cu același nume ca al obiectului șters, dar UID-urile trebuie să fie unice pe toată durata de viață a *Cluster*-ului. UID-urile sunt generate de Kubernetes.

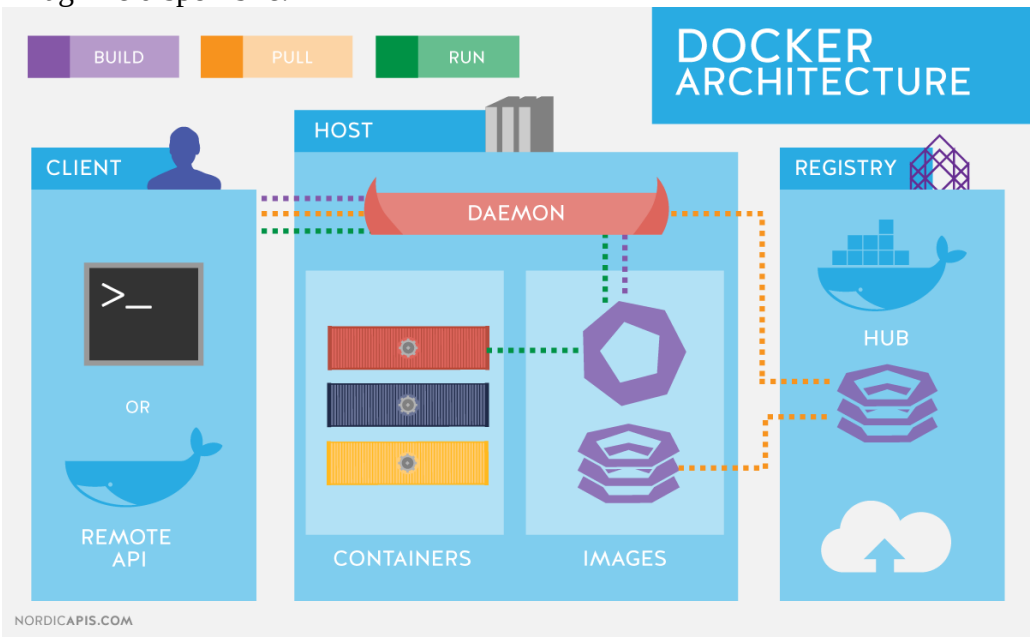
n) *Namespace* - un spațiu de nume este un *Cluster* virtual. Un *Cluster* fizic poate conține mai multe *Cluster*-e virtuale separate de spații de nume. Fiecare *Cluster* virtual este complet izolat de restul *Cluster*-elor virtuale și poate comunica numai prin intermediul interfeței publice. Nodurile și volumele persistente nu sunt într-un spațiu de nume. Kubernetes poate programa modulele din diferite spații de nume pentru a rula pe același nod. De asemenea, modulele din diferite spații de nume pot utiliza aceeași stocare persistentă.

Capitolul 4. Analiză şi Fundamentare Teoretică

4.1. Utilizarea Docker

4.1.1. Generalități

Arhitectura Docker utilizează un model client-server și cuprinde componentele Docker Client, Docker Host și registrul Docker Hub (Figura 4.1). Ca și în cazul oricărei alte platforme, primul pas pentru a rula o aplicație este obținerea fișierelor binare necesare. Pentru Docker, întreaga aplicație este ambalată într-o imagine și aceste imagini sunt stocate în registre. În prezent, există un registru implicit, care poate fi explorat folosind un browser sau folosind comanda de căutare Docker (Figura 4.2), care listează toate imaginile disponibile.



Figură 4.1 Fluxul de lucru în Docker

```
root@dan-VirtualBox:~# docker search tomcat
NAME                                DESCRIPTION                                     STARS               OFFICIAL            AUTOMATED
tomcat                              Apache Tomcat is an open source implementati…   1898                [OK]
tomee                               Apache TomEE is an all-Apache Java EE certif…   51                  [OK]
dordoka/tomcat                      Ubuntu 14.04, Oracle JDK 8 and Tomcat 8 base…   49                                      [OK]
davidcaste/alpine-tomcat            Apache Tomcat 7/8 using Oracle Java 7/8 with…   25                                      [OK]
bitnami/tomcat                      Bitnami Tomcat Docker Image                     17                                      [OK]
consol/tomcat-7.0                   Tomcat 7.0.57, 8080, "admin/admin"              16                                      [OK]
cloudesire/tomcat                   Tomcat server, 6/7/8                            15                                      [OK]
tutum/tomcat                        Base docker image to run a Tomcat applicatio…   10
jeanblanchard/tomcat                Minimal Docker image with Apache Tomcat         8
meirwa/spring-boot-tomcat-mysql-app a sample spring-boot app using tomcat and My…   8                                       [OK]
aallam/tomcat-mysql                 Debian, Oracle JDK, Tomcat & MySQL              7                                       [OK]
rightctrl/tomcat                    CentOS , Oracle Java, tomcat application ssl…   3                                       [OK]
amd64/tomcat                        Apache Tomcat is an open source implementati…   2
fabric8/tomcat-8                    Fabric8 Tomcat 8 Image                          2                                       [OK]
maluuba/tomcat7-java8               Tomcat7 with java8.                             2
camptocamp/tomcat-logback           Docker image for tomcat with logback integra…   1                                       [OK]
99taxis/tomcat7                     Tomcat7                                         1                                       [OK]
primetoninc/tomcat                  Apache tomcat 8.5, 8.0, 7.0                     1                                       [OK]
oobsri/tomcat8                      Testing CI Jobs with different names.           0
jelastic/tomcat                                                                     0
swisstopo/service-print-tomcat      backend tomcat for service-print "the true, …   0
s390x/tomcat                        Apache Tomcat is an open source implementati…   0
trollin/tomcat                                                                      0
picoded/tomcat7                     tomcat7 with jre8 and MANAGER_USER / MANAGER…   0                                       [OK]
awscory/tomcat                      tomcat                                          0
root@dan-VirtualBox:~# ~~~
```

Figură 4.2 *Comanda docker search*

Comanda de căutare redă de fapt depozite, care pot conține mai multe versiuni ale unei anumite imagini, fiecare având o etichetă de versiune diferită, dar doar una este etichetată ca fiind „cea mai recentă”. O imagine de Docker este identificată în mod unic printr-un ID. Numai ID-ul imaginii este un identificator cu adevărat unic, deoarece chiar dacă două imagini pot proveni din același depozit și au aceeași etichetă de versiune, ele pot fi diferite, de exemplu în cazul etichetei *latest*.

Scopul Docker este virtualizarea, astfel că, atunci când se execută o imagine, Docker creează un container și plasează aplicația din imagine în interiorul acestuia. Pentru a rula o imagine, se poate specifica fie ID-ul imaginii, numele depozitului cu etichetă, fie numai numele depozitului, caz în care se folosește eticheta implicită *latest*. In figura de mai jos se poat observa crearea unui container Tomcat cu comanda *docker run -t -i tomcat*.

```
root@dan-VirtualBox:~# docker run -t -i tomcat
Unable to find image 'tomcat:latest' locally
latest: Pulling from library/tomcat
0bd44ff9c2cf: Pull complete
047670ddbd2a: Pull complete
ea7d5dc89438: Pull complete
4a0557097lbb: Pull complete
66f679cd5859: Pull complete
89362eaac850: Pull complete
d76c23323cb4: Pull complete
f7all3d2d566: Pull complete
f7ffd00be2be: Pull complete
dd678d267c76: Pull complete
97f6f322fa52: Pull complete
5ld085dee99e: Pull complete
Digest: sha256:e99c57ccb609f64c332ee889ddl4c8fcc050dd65afed8bd0c0bc7ecele4ad2cd
Status: Downloaded newer image for tomcat:latest
```

Figură 4.3 Crearea unui container Tomcat

Parametrii *-t -i* provoacă Docker-ul să atașeze un pseudo TTY la container, astfel încât, după ce toate straturile sunt trase, poate fi urmărită consola containerului (Figura 4.4).

```
on directory [/usr/local/tomcat/webapps/examples]
28-Jun-2018 15:39:14.815 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web appli
cation directory [/usr/local/tomcat/webapps/examples] has finished in [294] ms
28-Jun-2018 15:39:14.816 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web applicati
on directory [/usr/local/tomcat/webapps/host-manager]
28-Jun-2018 15:39:14.868 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web appli
cation directory [/usr/local/tomcat/webapps/host-manager] has finished in [52] ms
28-Jun-2018 15:39:14.878 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
28-Jun-2018 15:39:14.916 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]
28-Jun-2018 15:39:14.926 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in 912 ms
```

Figură 4.4 *Log*-urile din containerul Tomcat

Se pot transmite și alți parametrii opționali, inclusiv opțiunile de denumire a containerului, specificarea mapării portului din container la gazdă sau utilizarea serverelor DNS personalizate pentru container.

În Figura 4.3 se poate observa că imaginea a fost descărcată automat din registru, înainte de a fi rulată. Dacă se dorește doar descărcarea imaginii se poate folosi comanda *docker pull*. O altă opțiune care ar putea fi dorită este doar crearea unui container dintr-o imagine, dar nu și rularea acestuia. În acest scop, există comanda *docker create*.

4.1.2. Containere

Pentru containerele din gazdă, Docker oferă toate caracteristicile de bază, cum ar fi listarea containerelor care rulează, oprirea sau eliminarea lor. Capacitățile mai

interesante sunt examinarea jurnalelor (*docker logs*), vizualizarea proceselor din interiorul containerelor (*docker top*) sau conectarea la un container care rulează (*docker attach*). Una dintre opțiunile importante este capacitatea de a rula containerul în fundal folosind modul *daemon* și parametrul *-d*.

La pornirea unui container, este posibil să se impună restricții privind numărul de resurse pe care le poate utiliza. Pentru a forța o limită maximă de memorie pentru un anumit container, se utilizează parametrul *-m* și parametrul *-c* pentru a da un nivel de prioritate CPU. Acești parametrii sunt de fapt transmiși modulului *cgroups*. Maparea porturilor containerului către porturile gazdei permite posibilitatea de a avea mai multe containere, în care procesele rulează pe același port.

4.1.3. Imagini

În mediul Docker, aplicațiile sunt furnizate sub formă de imagini de aplicație. O astfel de imagine conține aplicația împreună cu o referință la imaginea sa sursă, astfel încât se creează o arhitectură stratificată. Dacă o imagine nu are o imagine sursă, se numește imagine de bază sau rădăcină. Exemple comune de imagini rădăcină sunt cele ale distribuțiilor Linux.

În prezent, există două moduri prin care se poate crea o imagine de aplicație. Se poate rula o imagine existentă ca un container și se pot efectua modificări. Comanda *docker commit* salvează modificările și poate fi urmată de comanda *docker push,* stocând modificările într-un depozit. Cealaltă modalitate este de a profita de construirea automată a imaginilor. Acest lucru este gestionat de o comandă de construire a docker-ului (*docker build*), care necesită ca instrucțiunile să fie furnizate într-un fișier Docker (*Dockerfile*). Instrucțiunile acceptate includ copierea și descărcarea fișierelor, executarea de scripturi, specificarea imaginilor sursă și a volumelor de date, expunerea porturilor și alte câteva opțiuni. Este posibilă și crearea unei imagini de bază, fie prin furnizarea unei arhive *tar* a unui sistem de fișiere existent în comanda de import a docker-ului (*docker import*), fie prin specificarea unei imagini speciale numită zgârietură (conține un sistem de fișiere gol) ca imagine sursă.

O secvență de instrucțiuni Dockerfile pentru o aplicație poate să arate astfel:

```
FROM repository:image              #imagine sursă
MAINTAINER name email              #cel care se ocupă de imagine
ENV key value                      #setarea variabilelor mediului
RUN/ADD/COPY commands              #instalare aplicații, copiere și descărcare de date
EXPOSE port1, port2                #expunere porturi
ENTRYPOINT /path/to/app            #calea spre aplicație
VOLUME /path/to/data/volume        #volum de date
```

4.1.4. Registre

Există registre publice, Docker Hub, sau registre private. În interiorul registrului, diferitele imagini ale unei aplicații se află într-un depozit. Utilizarea unui registru este similară cu utilizarea sistemului de control *Git*. Comenzile *push* și *pull* sunt folosite pentru a interacționa cu registrul.

Registrele au câteva funcții care ajută la crearea de noi versiuni ale imaginilor. Construcțiile automate oferă o modalitate de a construi automat o imagine de aplicație într-un depozit dintr-o sursă *github* sau *bitbucket* și dintr-un fișier Docker, în timp ce

lanțurile Webhooks și Webhook permit trimiterea uneia sau mai multor cereri HTTP cu payload JSON. Acesta poate fi folosit ca mod de a trimite notificări despre o nouă actualizare a imaginii aplicației.

În prezent există două versiuni ale proiectului de registru Docker, ambele fiind open source. Cel mai vechi (v1), scris în Python, a fost folosit în Docker până în versiunea 1.6, după care, în versiunea curentă, Distribution (v2), a fost scrisă în Go. Distribuția pretinde să ofere cereri *push* și *pull* mai rapide, și o implementare mai eficientă. Chiar dacă se încearcă menținerea compatibilității cu versiunile anterioare, unele *endpoint-uri* ale registrului variază puțin.

4.1.5. Docker Hub

Diferențele majore dintre Docker Hub și un registru constau în faptul că există o singură instanță a Docker Hub (gestionată de compania Docker) și gestionează autentificarea și autorizarea utilizatorilor și conține sumele de control ale imaginilor, în timp ce pot exista mai multe registre care stochează imagini Docker. Docker Hub găzduiește cel mai mare registru public, astfel încât uneori termenii Docker Hub și registru public sunt interschimbate.

Registrul Docker Hub conține trei tipuri de depozite: depozite oficiale, care conțin imagini de la furnizori și contribuitori Docker; depozite private, unde pot fi păstrate imagini non-publice; depozite publice pentru schimbul de imagini. Începând cu aprilie 2015, registrul public al Docker Hub a oferit peste 45 000 de imagini, ceea ce înseamnă că toate programele Linux majore sunt deja construite și publicate

4.1.6. Docker Swarm

Docker Swarm este un instrument de grupare pentru Docker. Cu alte cuvinte, el ia mai multe motoare Docker și le expune ca o singură instanță. Toate comenzile standard Docker sunt disponibile, dar au mai fost adăugate câteva, pentru a configura grupul și a specifica plasarea containerelor. Există mai multe opțiuni de adăugare a unor mașini individuale în *Cluster*. Soluția cea mai dinamică este generarea unui id de *Cluster* și apoi folosirea acestuia în comanda *swarm join* în fiecare nod. Alte metode ar fi: atașarea unei liste de adrese IP ale nodurilor în mod explicit sau transmiterea unui nume de fișier pentru a încărca lista din el.

După formarea grupului, o comandă *swarm manage* va porni managerul de grupuri și comenzile Docker vor fi emise pentru întregul grup. La crearea de containere noi, trebuie selectată o strategie pentru a decide atribuirea nodurilor. Strategia implicită, BinPacking, încearcă să evite fragmentarea și, prin urmare, plasează noul container în nodul cu cea mai mare utilizare a resurselor, care are în continuare suficiente resurse pentru a-l rula. A doua strategie implementată în prezent este de a alege pur și simplu un nod aleatoriu.

Când se utilizează Docker Swarm, pot fi transmiși parametrii suplimentari comenzii *docker run*, care poate adăuga restricții suplimentare pentru alegerea nodului pe care să ruleze imaginea. Astfel de restricții se numesc filtre și pot forța sau exclude selecția unui anumit nod după nume, porturile disponibile, sistemul de operare, versiunea kernel, tipul de stocare a nodului sau plasarea containerului pe același nod ca un alt container deja plasat.

4.1.7. Docker Machine

Utilitarul Docker Machine este dovada faptului că virtualizarea clasică și cea containerizată pot coexista și se pot complementa. Se folosește o mașină virtuală dedicată a cărei funcție principală este de a rula Docker Daemons. De asemenea, se poate integra cu Docker Swarm, astfel fiind creat și un *Cluster*.

Docker Machine oferă toate comenzile de bază pentru gestiune, astfel se pot crea, porni, reporni, opri, elimina și lista mașinile virtuale, dar se poate și actualiza Docker Daemons sau conecta prin SSH la o mașină virtuală. Această abordare oferă mai multe avantaje decât folosirea Docker Daemons. În primul rând, Docker Machine nu este disponibil numai pe Linux, ci și pentru Windows și OS X. În al doilea rând, acesta poate funcționa cu ambele mașini virtuale locale, cum ar fi cele create de VirtualBox sau cu soluții cloud (Microsoft Azure). Ultima, dar nu cea din urmă, Docker Machine își propune să aibă grijă de sarcini comune cum ar fi generarea de chei SSH pentru noile mașini sau instalarea Boot2Docker ca sistem de operare pe o mașină nou creată.

4.1.8. Docker Compose

Un Docker Compose (software complex) este compus din mai multe componente, cum ar fi un server web și o bază de date. Instalarea manuală a fiecărei componente ar fi o sarcină repetitivă, motiv pentru care a fost creat proiectul *Fig*. În februarie 2015, acesta a fost depreciat în favoarea Docker Compose, care se bazează pe codul proiectului *Fig*.

Ideea principală este capacitatea de a conecta mai multe containere, permițând să se compună o aplicație complexă din mai multe imagini. Utilizează o configurație YAML (Ain't Markup Language, limbaj de serializare a datelor care poate fi citit de om), care permite specificarea configurației pentru expunerea porturilor, transmiterea variabilelor de mediu și montarea volumelor de date. O configurație poate extinde o altă configurație, de exemplu, o aplicație poate avea o configurație simplă și două complexe: una pentru dezvoltare și cealaltă pentru producție. Docker Compose funcționează bine împreună cu Docker Swarm, unde containerele legate sunt programate pe aceeași gazdă.

4.2. Utilizarea Kubernetes

Pentru a realiza toate funcționalitățile incredibile pe care le are, Kubernetes are o rețea de componente, legate între ele, care lucrează continuu pentru a crea starea dorită a grupurilor de containere. Toate componentele monitorizează starea curentă și încearcă să o sincronizarea cu starea dorită. Folosirea acestor componente este descrisă în continuare. Descrierea componentelor se face în YAML (*Ain't Markup Language),* un limbaj de serializare a datelor care poate fi citit cu ușurință de oameni. Este utilizat în mod obișnuit pentru fișierele de configurare, dar ar putea fi utilizat în multe aplicații în care sunt stocate sau transmise date.

4.2.1. Pod

Fișier YAML care descrie un *Pod* numit *nginx-pod,* cu ultima imagine *nginx,* care rulează pe portul 80 este prezentat în Figura 4.5. Acesta se crează cu comanda *kubectl create -f nginx-pod.yaml*, se poate obține cu comanda *kubectl get pods* și se poate observa descrierea lui cu comanda *kubectl describe pods/nginx-pod* (Figura 4.6). IP-ul *Pod*-ului este unul privat, deci nu poate fi accesat direct de pe mașina locală. Însă, *kubectl exec*

folosește funcționalitatea *docker exec*, care are accesul necesar (*kubectl exec nginx-pod -- curl <adresă ip privată>*).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - image: nginx:latest
    name: nginx-pod
    ports:
    - containerPort: 80
```

Figură 4.5 *nginx-pod.yaml*

```
>kubectl describe pods/nginx-pod
Name:         nginx-pod
Namespace:    default
Node:         minikube/192.168.99.100
Start Time:   Fri, 29 Jun 2018 19:07:30 +0300
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           172.17.0.7
Containers:
  nginx-pod:
    Container ID:   docker://0614bf8c9c8bd2585868bc4c2afe46d47994a16cb021ae9bebad3c56d7d3d7b3
    Image:          nginx:latest
    Image ID:       docker-pullable://nginx@sha256:62a095e5da5f977b9f830adaf64d604c614024bf239d21068e4ca826d0d629a4
    Port:           80/TCP
    State:          Running
      Started:      Fri, 29 Jun 2018 19:07:35 +0300
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-n85hq (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-n85hq:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-n85hq
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
```

Figură 4.6 Descrierea Pod-ului *nginx-pod*

4.2.2. Replication Controller

Descrierea unui Replication Controller (RC) *nginx* (Figura 4.7):

- *Kind*: acesta specifică tipul de resursă pe care o creăm. În acest caz, tipul este ReplicationController. Scriptul *kubectl* utilizează o singură comandă de creare pentru toate tipurile de resurse. Beneficiul este că se pot crea cu ușurință o serie de resurse de diferite tipuri fără a fi nevoie specificarea de parametrii individuali pentru fiecare tip. Cu toate acestea, este necesar ca fișierele de definiție să poată identifica tipul de resursă pe care îl specifică.
- *apiVersion*: acest lucru specifică care versiune a schemei se folosește.
- *Metadata*: atribuie resurselor un nume și specifică etichetele care vor fi folosite pentru a căuta și selecta resursele pentru o anumită operație. Metadatele permit și crearea de observații pentru informațiile care nu pot fi

identificate și care ar putea fi utile pentru instrumentele și bibliotecile clientului.

- *Spec*: variază în funcție de tipul de resurse create. În acest caz, este ReplicationController, care asigură numărul dorit de *Pod*-uri care rulează. Elementul *replicas* definește numărul dorit de *Pod*-uri, elementul selector îi spune controlerului *Pod*-urile care vor fi urmărite, iar elementul *template* definește un șablon pentru lansarea unui nou *Pod*. Secțiunea *template* conține aceleași detalii ca în definiția *Pod*-ului. Un lucru important este faptul că valorile selectorului trebuie să se potrivească cu valorile de etichete specificate în șablonul *Pod*-ului. Această potrivire este utilizată pentru a selecta *Pod*-urile administrate.

Crearea RC-ului se face cu *kubectl create -f nginx-controller.yaml* și verificarea acestuia se face cu *kubectl get rc* (Figura 4.8). Cu comanda *kubectl get pods* (Figura 4.9) putem să observăm Pod-urile create de acest RC, denumite după numele RC-ului, urmat de un șir aleator de caractere. Aici se mai poate vedea și status-ul sau numărul de restartări ale fiecărui Pod.

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  replicas: 3
  selector:
    name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Figură 4.7 *nginx-controller.yaml*

```
>kubectl create -f nginx-controller.yaml
replicationcontroller "nginx" created

>kubectl get rc
NAME      DESIRED   CURRENT   READY     AGE
nginx     3         3         3         1m
```

Figură 4.8 Crearea și vizionarea RC-ului

```
>kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
myapp-7b99bfdcbf-9jlwg  1/1       Running   1          9d
mysql-7d67dbb457-cp7ms  1/1       Running   2          10d
mytom-557d89c5c5-bghkz  1/1       Running   2          10d
nginx-knp4l             1/1       Running   0          1m
nginx-n7nzh             1/1       Running   0          1m
nginx-q48c5             1/1       Running   0          1m
```

Figură 4.9 Pod-urile RC-ului

4.2.3. Service

Descrierea YAML a unui serviciu este similară cu cea a unui Replication Controller (Figura 4.10). Principala diferență se poate observa în elementul *spec* al serviciului. Se definește tipul de serviciu (*LoadBalancer*), portul (*80*) și selectorul (*name: nginx*), care indică proxy-ului care *Pod*-uri pot răspunde serviciului.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    name: nginx
```

Figură 4.10 *nginx-service.yaml*

Pe Google Kubernetes Engine (GKE), aceasta va crea un *Load Balancer* extern și reguli de redirecționare, dar este posibil să fie nevoie de reguli suplimentare de *firewall*. *Firewall*-ul portului 80 și 443 este deschise implicit. Astfel, poate fi necesară deschiderea port-ului dacă se folosește un serviciu cu alte port-uri decât 80 și 443.

Verificarea Serviciului se poate observa în Figura 4.11. Dupa crearea serviciului, atribuirea ip-ul extern este în desfașurare.

```
>kubectl create -f nginx-service.yaml
service "nginx" created

>kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP          116d
myapp        NodePort       10.109.89.134   <none>        4200:31586/TCP   10d
mysql        ClusterIP      10.98.68.77     <none>        3306/TCP         11d
mytom        NodePort       10.105.98.174   <none>        8080:30180/TCP   11d
nginx        LoadBalancer   10.110.93.78    <pending>     80:32575/TCP     <invalid>
```

Figură 4.11 Crearea și verificarea serviciului nginx

În descrierea serviciului (Figura 4.12) se pot observa mai multe elemente cheie. *Namespace*-ul este setat implicit, tipul este LoadBalancer și IP-ul extern ar trebui să fie afișat sub *LoadBalancer Ingress*, dar este folosit Minikube, care nu atribuie IP extern. *Endpoint-urile* arată IP-urile *Pod*-urilor disponibile care pot răspunde la solicitările serviciului.

```
>kubectl describe service/nginx
Name:                     nginx
Namespace:                default
Labels:                   name=nginx
Annotations:              <none>
Selector:                 name=nginx
Type:                     LoadBalancer
IP:                       10.110.93.78
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32575/TCP
Endpoints:                172.17.0.7:80,172.17.0.8:80,172.17.0.9:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Figură 4.12 Descrierea serviciului *nginx*

Tipurile de servicii disponibile sunt:

- *ClusterIP*: Expune serviciul pe un IP intern de tip *Cluster*. Alegerea acestei valori face ca serviciul să fie accesibil numai din interiorul *Cluster*-ului. Acesta este serviciul prestabilit.
- *NodePort*: expune serviciul pe IP-ul fiecărui nod, la un port static (*NodePort*). Un serviciu *ClusterIP*, la care se va direcționa serviciul *NodePort*, este creat automat. Serviciul NodePort se poate apela, din afara *Cluster*-ului, solicitând <*NodeIP*>:<*NodePort*>.
- *LoadBalancer*: Expune serviciul extern utilizând încărcarea echilibrată a unui furnizor de cloud. Serviciile *NodePort* și *ClusterIP*, la care se va direcționa echilibrarea externă, sunt create automat.
- *ExternalName*: Mapează serviciul la conținutul câmpului *externalName* (de ex. dan.licenta.com), returnând o înregistrare CNAME cu valoarea sa. Nu există niciun fel de proxy. Aceasta necesită versiunea 1.7 sau o versiune mai mare a *kube-dns*.

4.2.4. Deployment

Începând cu versiunea 1.2, Kubernetes a adăugat constructorul Deployment, care îmbunătățește mecanismele de bază ale actualizării în timpul rulării (*rolling-update*) și ale controlorilor de replicare (Replication Controllers). După cum sugerează și numele, acesta oferă un control mai bun asupra implementării codului în sine. Deployment-urile ne permit întreruperea și reluarea aplicațiilor. În plus, păstrează un istoric al Deployment-urilor și permite revenirea la versiunile anterioare.

În Figura 4.13 se observă că definiția unui Deployment este foarte asemănătoare cu cea a unui Replication Controller. Principala diferență este capacitatea de a face schimbări și actualizări ale obiectelor Deployment, în timp ce Kubernetes gestionează actualizarea *Pod*-urilor și a replicilor automat.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    name: nginx-deploy
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: nginx-deploy
    spec:
      containers:
      - name: nginx-deploy
        image: nginx:alpine
        ports:
        - containerPort: 80
```

Figură 4.13 *nginx-deploy.yaml*

Crearea unui Deployment se face cu comanda *kubectl create -f nginx-deploy.yaml --record*, unde *--record* este opțional și provoacă salvarea acestuia în istoricul de *rollout*.

Verificarea se face cu comanda *kubectl get pods -l name=nginx-deploy* (ilustrată în Figura 4.14), parametrul *-l* fiind folosit pentru a vedea doar *Pod*-urile Deployment-ului.

```
>kubectl create -f nginx-deploy.yaml --record
deployment "nginx-deploy" created

>kubectl get pods -l name=nginx-deploy
NAME                            READY     STATUS    RESTARTS   AGE
nginx-deploy-6464bf9d7f-rj5hl   1/1       Running   0          <invalid>
```

Figură 4.14 Crearea și verificarea *Deployment*-ului *nginx*

4.2.5. Scaling

Unele aplicații au nevoie de mai multe resurse, în timp ce altele pot funcționa cu mai puține resurse. Din fericire, Kubernetes include comanda *scale*, care funcționează atât cu Replication Controller (*RC*), cât și cu un Deployment. Comanda pentru scalarea RC este *kubectl scale --replicas=3 rc/nginx-rc.yaml*, iar cea pentru Deployment este *kubectl scale deployment nginx-deploy --replicas 3*. După execuție se pot verifica *Pod*-urile corespunzătoare Deployment-ului, care ar trebui să fie 3, după cum se poate oberva și în Figura 4.15. Dacă se analizează *Pod*-urile, se poate vedea că primul și ultimul *Pod* sunt *Pod*-urile noi create din cauza campului *AGE*, care indică că cele două sunt proaspăt create și că cel din mijloc este mai vechi.

```
>kubectl scale deployment nginx-deploy --replicas 3
deployment "nginx-deploy" scaled

>kubectl get pods -l name=nginx-deploy
NAME                            READY     STATUS    RESTARTS   AGE
nginx-deploy-6464bf9d7f-cbxvr   1/1       Running   0          <invalid>
nginx-deploy-6464bf9d7f-rj5hl   1/1       Running   0          23m
nginx-deploy-6464bf9d7f-zmkvm   1/1       Running   0          <invalid>
```

Figură 4.15 Scalarea *Deployment*-ului *nginx*

De asemenea, comanda poate fi utilizată și pentru a reduce numărul de replici. În ambele cazuri, comanda de scalare adaugă sau elimină replicile *Pod* necesare, iar serviciul se actualizează automat și echilibrează traficul pe replicile noi sau pe cele rămase.

4.2.6. Smooth Updates

Scalarea aplicației în sus și în jos după cerințele resurselor este utilă pentru multe scenarii de producție, dar ce se poate face în cazul actualizărilor simple de aplicații? Orice sistem de producție are actualizări ale codului, corecții și adăugări. Acestea ar putea apărea lunar, săptămânal sau chiar zilnic. Asigurarea unei modalități fiabile de a face aceste schimbări fără întreruperi este un aspect important.

Există un suport integrat pentru rularea actualizărilor cu versiunea 1.0. Comanda *rolling-update* permite actualizarea tuturor replicilor sau doar imaginea Docker folosită de fiecare replică. De asemenea, se poate specifica un interval de actualizare, care va permite actualizarea *Pod*-urilor unul câte unul, care presupune crearea unui nou Pod, așteptarea intervalului de actualizare, ștergerea unui Pod vechi, după care se ia de la început cu crearea unui alt Pod nou, până când nu mai există Pod-uri vechi.

Actualizarea în versiunea *perl* a imaginii containerului *nginx* (*RC*), la un interval de actualizare de 2 minute se face cu comanda *kubectl rolling-update nginx nginx-perl --image=nginx:perl --update-period=”2m”*, unde *nginx-perl* este numele noului RC, în lipsa căruia se folosește același nume urmat de un cod lung (de exemplu *nginx-b74649c315b380ef4640e84bddab9d48-mlp98*). În Figura 4.16 se poate observa cum se crează un nou *Pod, nginx-perl*, după care se șterge un *Pod nginx*. În Figura 4.17 sunt vizibile *Pod*-urile în momentul în care se execută *Scaling nginx down to 0*, având 3 *Pod*-uri actualizate (*nginx-perl*) și un *Pod* care este șters (*nginx*).

```
>kubectl rolling-update nginx nginx-perl --image=nginx:perl --update-period="2m
"
Created nginx-perl
Scaling up nginx-perl from 0 to 3, scaling down nginx from 3 to 0 (keep 3 pods
available, don't exceed 4 pods)
Scaling nginx-perl up to 1
Scaling nginx down to 2
Scaling nginx-perl up to 2
Scaling nginx down to 1
Scaling nginx-perl up to 3
Scaling nginx down to 0
Update succeeded. Deleting nginx
replicationcontroller "nginx" rolling updated to "nginx-perl"
```

Figură 4.16 Rolling update

```
>kubectl get pods -l name=nginx
NAME                READY     STATUS        RESTARTS   AGE
nginx-n7nzh         0/1       Terminating   1          5d
nginx-perl-8fxgk    1/1       Running       0          54s
nginx-perl-dt98f    1/1       Running       0          4m
nginx-perl-fgbxz    1/1       Running       0          2m
```

Figură 4.17 Pod-urile în momentul în care este șters ultimul Pod vechi

Merită remarcat faptul că, pe parcursul întregului proces de actualizare, doar *Pod*-urile și RC-urile au fost afectate. Serviciul a rămas funcțional în continuare, cu direcționarea la noua versiune a *Pod*-urilor. Acest lucru se datorează faptului că serviciul folosește selectori de etichete pentru alegerea membrilor. Deoarece replicile vechi și cele noi utilizează aceleași etichete, serviciul nu are nici o problemă în utilizarea noilor *Pod*-uri pentru a răspunde solicitărilor. Actualizările se fac pe *Pod*-uri unul câte unul, astfel încât serviciul poate să răspundă solicitărilor clienților.

Deployment-urile permit actualizarea în câteva moduri diferite. Mai întâi, există comanda *kubectl set*, care permite actualizarea imaginii. Actualizarea imaginii Deployment-ului *nginx-deploy* la *nginx:perl* se face cu comanda *kubectl set image deployment/nginx-deploy nginx-deploy=nginx:perl* și se poate observa în Figura 4.18. Verificarea acestei comenzi se poate face cu *kubectl rollout status deployment/nginx-deploy*, care ar trebui să returneze *deployment "nginx-deploy" successfully rolled out*. În Figura 4.19 se poate observa că *Pod*-urile Deployment-ului sunt recreate de comanda *kubectl set image*.

```
PS C:\Users\danutz13s> kubectl describe deployment nginx-deploy | Select-String -Pattern "Image:"

    Image:        nginx:alpine


PS C:\Users\danutz13s> kubectl set image deployment/nginx-deploy nginx-deploy=nginx:perl
deployment "nginx-deploy" image updated
PS C:\Users\danutz13s> kubectl describe deployment nginx-deploy | Select-String -Pattern "Image:"

    Image:        nginx:perl
```

Figură 4.18 Setarea imaginii *nginx:perl* pentru Deployment-ul *nginx*

```
>kubectl get pods -l name=nginx-deploy
NAME                            READY     STATUS    RESTARTS   AGE
nginx-deploy-dfddf7889-q7dcj    1/1       Running   0          19m
nginx-deploy-dfddf7889-vtdq9    1/1       Running   0          19m
nginx-deploy-dfddf7889-wfhjg    1/1       Running   0          19m

>kubectl set image deployment/nginx-deploy nginx-deploy=nginx:alpine
deployment "nginx-deploy" image updated

>kubectl get pods -l name=nginx-deploy
NAME                            READY     STATUS    RESTARTS   AGE
nginx-deploy-6464bf9d7f-65866   1/1       Running   0          <invalid>
nginx-deploy-6464bf9d7f-g4pq9   1/1       Running   0          <invalid>
nginx-deploy-6464bf9d7f-tsnzt   1/1       Running   0          <invalid>
```

Figură 4.19 Pod-urile Deployment-ului *nginx* înainte și după schimbarea imaginii

În spatele scenei, Kubernetes lansează o nouă versiune a Deployment-ului. În principiu, acesta creează un nou set de replici cu noua versiune. Odată ce un *Pod* nou este gata, acesta șterge una dintre versiunile mai vechi. Kubernetes continuă acest comportament, scalând în sus versiuni noi și în jos versiuni vechi, până când rămân doar noile *Pod*-uri.

Definiția *rollback*-ului permite controlarea metodei de înlocuire a unui *Pod* în definiția Deployment-ului. Există un câmp *strategy.type* care este implict *RollingUpdate*. Opțional, se poate specifica și recrearea (*Recreate*) ca strategie de înlocuire, care va șterge toate *Pod*-urile vechi înainte de a crea noile versiuni.

Istoricul schimbărilor poate fi obținut cu comanda *kubectl rollout history deployment/nginx-deploy*, care poate fi vizualizată și în Figura 4.20. Pe lângă status și istoric, comanda acceptă și *pause*, *resume* și *undo*. Comanda *pause* permite întreruperea unei comenzi în timp ce *rollout*-ul este încă în desfășurare. Acest lucru poate fi util pentru rezolvarea problemelor și pentru lansările de tip *canary*, unde se efectuează testarea finală a noii versiuni înainte de lansarea către întreaga bază de utilizatori. Când se poate continua lansarea, se folosește comanda *resume*.

```
>kubectl rollout history deployment/nginx-deploy
deployments "nginx-deploy"
REVISION  CHANGE-CAUSE
4         kubectl.exe set image deployment/nginx-deploy nginx-deploy=nginx:perl
5         kubectl set image deployment/nginx-deploy nginx-deploy=nginx:alpine
```

Figură 4.20 Istoricul schimbărilor Deployment-ului *nginx*

Comanda *rollout undo* este folositoare când o schimbare provoacă probleme. De exemplu, dacă se setează imaginea cu o imagine care nu există, *Pod*-urile eșuează. Folosind comanda *kubectl rollout undo deployment/nginx-deploy* se face *rollback* la

versiunea anterioară. Se poate observa în Figura 4.21 că imaginea care nu exista este penultima din istoric, iar ultima este versiunea anterioară acesteia.

```
>kubectl set image deployment/nginx-deploy nginx-deploy=nginx:18
deployment "nginx-deploy" image updated

>kubectl rollout status deployment/nginx-deploy
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

>kubectl get pods -l name=nginx-deploy
NAME                            READY     STATUS             RESTARTS   AGE
nginx-deploy-6464bf9d7f-65866   1/1       Running            0          9h
nginx-deploy-6464bf9d7f-tsnzt   1/1       Running            0          9h
nginx-deploy-77578d7ccc-8kvbk   0/1       ImagePullBackOff   0          6s
nginx-deploy-77578d7ccc-sdv4k   0/1       ErrImagePull       0          6s

>kubectl rollout undo deployment/nginx-deploy
deployment "nginx-deploy"

>kubectl rollout history deployment/nginx-deploy
deployments "nginx-deploy"
REVISION  CHANGE-CAUSE
4         kubectl.exe set image deployment/nginx-deploy nginx-deploy=nginx:perl
6         kubectl set image deployment/nginx-deploy nginx-deploy=nginx:18
7         kubectl set image deployment/nginx-deploy nginx-deploy=nginx:alpine
```

Figură 4.21 Revenirea la imaginea anterioară a Deployment-ului *nginx*

4.2.7. Autoscaling

O caracteristică Kubernetes este Horizontal Pod Autoscaler-ul (HPA). Acest tip de resursă este utilă deoarece oferă o modalitate de a seta automat pragurile pentru a scala aplicația. Descrierea acestuia pentru Deployment-ul *nginx-deploy*, cu minim 3 replici și maxim 6 replici, cu pragul de utilizare al procesorului 10 se poate observa în Figura 4.22. Se poate executa și cu comanda *kubectl autoscale deploy nginx-deploy --min=3 --max=6 --cpu-percent=10*. Crearea HPA-ului se face cu comanda *kubectl create -f nginx-hpa.yaml*, după care se verifică cu *kubectl get hpa nginx-deploy* (Figura 4.23).

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-deploy
spec:
  maxReplicas: 6
  minReplicas: 3
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: nginx-deploy
  targetCPUUtilizationPercentage: 10
```

Figură 4.22 *nginx-hpa.yaml*

```
>kubectl create -f nginx-hpa.yaml
horizontalpodautoscaler "nginx-deploy" created

>kubectl get hpa nginx-deploy
NAME           REFERENCE                 TARGETS           MINPODS   MAXPODS   REPLICAS   AGE
nginx-deploy   Deployment/nginx-deploy   <unknown> / 10%   3         6         0          <invalid>
```

Figură 4.23 Crearea și verificarea HPA-ului

4.2.8. Persistent Storage

Aplicațiile poartă adesea date de stare și de înregistrare care nu pot fi pierdute. Caracterul tranzitoriu al containerelor poate fi o mare provocare. Când containerul este șters, datele dispar o dată cu el. Același lucru este valabil și pentru containerele eșuate pe care Kubernetes le repornește. Pentru acest lucru sunt folosite volumele sau discurile.

Un volum care există în afara containerului permite salvarea datelor importante pentru a fi disponibile după eventualele eșuări sau restartări. Mai mult, dacă avem un volum la nivel de *Pod*, datele pot fi partajate între containerele din aceeași stivă de aplicații și din același *Pod*. Docker însuși sprijină volumele, dar Kubernetes oferă o stocare persistentă, care rezistă mai mult decât durata de viață a unui singur container. Volumele sunt legate de *Pod*-uri, depind în totalitate de ele. În plus, un *Pod* poate avea mai multe volume, dintr-o varietate de surse.

Una dintre cele mai ușoare căi de a obține o persistență pe fondul eșuărilor de containere și a schimbului de date într-un *Pod* este utilizarea volumul *emptydir*. Acest tip de volum poate fi folosit fie cu volumele de stocare ale mașinii nodului în sine, fie cu un disc RAM opțional pentru o performanță mai mare.

Persistența este îmbunătățită dincolo de un singur container, dar când se elimină un *Pod*, datele vor fi pierdute. De asemenea, restartarea mașinii va șterge orice date din discurile RAM. Utilizarea unui disc temporar, în memoria RAM, se poate observa în Figura 4.24. După crearea *Pod*-ului se poate executa comanda *kubectl exec memory-pd – ls -lh | Select-String -Pattern "memory-pd"* pentru a obține directorul *memory-pd*, ca în Figura 4.25.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-pd
spec:
  containers:
  - image: nginx:latest
    ports:
    - containerPort: 80
    name: memory-pd
    volumeMounts:
    - mountPath: /memory-pd
      name: memory-volume
  volumes:
  - name: memory-volume
    emptyDir:
      medium: Memory
```

Figură 4.24 *temp.yaml*

```
PS C:\Users\danutz13s> kubectl create -f temp.yaml
pod "memory-pd" created
PS C:\Users\danutz13s> kubectl exec memory-pd -- ls -lh | Select-String -Pattern "memory-pd"

drwxrwxrwt   2 root root   40 Jul  5 10:07 memory-pd
```

Figură 4.25 Crearea și verificarea volumului temporar

Pentru a avea un spațiu de stocare gestionat separat, este nevoie de o modalitate pentru ca aplicația să specifice și să solicite stocarea fără a fi preocupată de modul în care este furnizată aceasta. Kubernetes oferă PersistentVolumes (PV) și PersistentVolumeClaims (PVC), care au fost concepute pentru această funcționalitate. PV

sunt similare cu volumele, dar sunt furnizate de administratorul *Cluster*-ului și nu depind de un anumit *Pod*. Acest volum poate fi revendicat de *Pod*-uri folosind PersistentVolumeClaims.

PersistentVolumeClaims permit specificarea detaliilor spațiului de stocare necesar. Se pot defini cantitatea spațiului de stocare și tipul de acces, cum ar fi ReadWriteOnce (citit și scris de un singur nod), ReadOnlyMany (citit de mai multe noduri) și ReadWriteMany (citit și scris de mai multe noduri). În Figura 4.26 este descris un PV cu acces ReadWriteOnce și 20Gi stocare, și PVC-ul care îl folosește. Figura 4.27 prezintă cele două componente și se poate observa că volumul PVC-ului este chiar volumul *pv-volume*.

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

Figură 4.26 *storage.yaml*

```
>kubectl create -f storage.yaml
persistentvolume "pv-volume" created
persistentvolumeclaim "pv-claim" created

>kubectl get pv pv-volume
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    CLAIM             STORAGECLASS   REASON    AGE
pv-volume   20Gi       RWO            Retain           Bound     default/pv-claim   manual                   <invalid>

>kubectl get pvc pv-claim
NAME       STATUS    VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pv-claim   Bound     pv-volume   20Gi       RWO            manual         <invalid>
```

Figură 4.27 Crearea și verificarea PV-ului și a PVC-ului

În plus, Kubernetes oferă alte două metode pentru specificarea anumitor grupări sau tipuri de volume de stocare. Primul este utilizarea selectorilor ca în cazul selecției *Pod*-urilor. Aici, etichetele pot fi aplicate volumelor de stocare, iar revendicările pot face referire la aceste etichete pentru a filtra volumele furnizate. În al doilea rând, Kubernetes are conceptul de StorageClass (SC) care permite specificarea unui furnizor de stocare și a parametrilor pentru tipul de volum pe care îl furnizează. Acesta permite accesarea furnizorului Google Cloud Persistent Disk. Comunitatea Kubernetes construiește

furnizori pentru o varietate de SC. Fiecare furnizor are propriul set de parametrii disponibili. Furnizorii mai permit și alegerea tipul de disc, precum și zona în care este necesar pentru a fi disponibil *Pod*-ului care se atașează la acesta. În Figura 4.28 este descris un StorageClass pentru Google Compute Engine.

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zones: us-central1-a, us-central1-b
  replication-type: none
```

Figură 4.28 Descriere YAML a unui StorageClass

Capitolul 5. Proiectare de Detaliu și Implementare

Pentru implementare am ales să virtualizez local o aplicație, într-un *Cluster* cu un singur nod, creat de Minikube. Crearea de *Cluster*-e pe plan local este importantă în timpul dezvoltării și în încercarea de a depana problemele la nivel local. Kubernetes este proiectat pentru aplicații cloud (aplicații care rulează în cloud). Astfel, am ales să pornesc aplicația și pe soluțiile containerizate oferite de furnizorii de cloud, cum ar fi Google Kubernetes Engine sau Azure Kubernetes Service.

5.1. Aplicația pentru virtualizare

Aplicația are ca temă un site de închirieri mașini. Pentru accesarea acestuia este necesară crearea unui cont de utilizator, care oferă posibilitatea de a vizualiza mașinile puse la închiriere într-o listă, detaliile acestora fiind accesibile prin apăsarea butonului de la baza anunțului. Închirierea unei mașini este posibilă pe zile. După rezervarea unei mașini se transmite un email cu datele rezervării si cu datele mașinii. După returnarea mașinii se transmite un alt email de mulțumire pentru utilizarea serviciilor puse la dispoziția utilizatorilor în care se specifică un link unde se poate acorda o notă mașinii folosite. Mașinile sunt gestionate de administratori.

Aplicația are ca temă o aplicație Web pentru închirierea mașinilor. Apăsând pe butonul *Account* din partea dreaptă-sus, este posibilă crearea unui cont de utilizator sau logarea într-un cont deja existent. Există două tipuri de conturi, cont utilizator normal, care poate viziona și închiria mașini, și cont pentru administrare, care poate crea, modifica și șterge mașini, operații vizibile doar acestuia. Mașinile sunt expuse pentru vizualizare într-o listă, unde sunt afișate și toate detaliile acestora. Închirierea unei mașini se face pe zile, după ce s-a apăsat butonul din partea de jos a anunțului. Zilele dorite sunt selectate din calendarul care apare după apăsarea butonului, iar închirierea este finalizată prin apăsarea butonului *Rent*. Dacă utilizatorul nu este logat, în locul butonului este afișat un mesaj corespunzător. După rezervarea unei mașini se transmite un email cu datele rezervării si cu datele mașinii. In momentul în care o mașină este ridicată de client, administratorul folosește butonul *Start* de pe pagina de închiriere pentru a specifica acest lucru. La returnarea mașinii, realizată de administrator prin apăsarea butonului *End* aflat pe pagina de închiriere, un alt email se transmite pentru mulțumirea utilizării serviciilor puse la dispoziția utilizatorilor, în care se specifică și un link pentru a acorda o notă mașinii folosite.

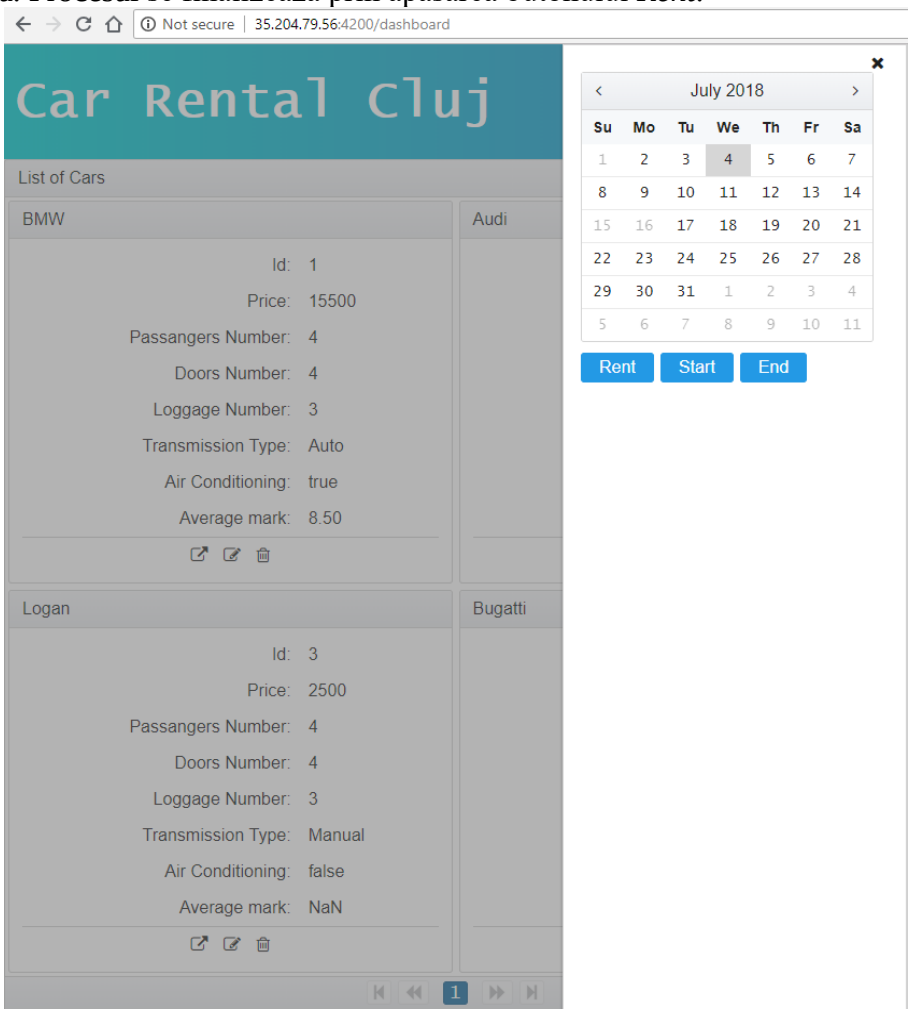
Baza de date MySql conține tabelul pentru stocarea utilizatorilor, care are un câmp special care verifică dacă un utilizator este administrator. Tabelul principal din baza de date este cel al mașinilor, care conține detalii precum numele, prețul pe zi, numărul de uși, numărul de persoane, numărul de bagaje, tipul de transmisie sau prezența aerului condiționat. Un alt tabel este cel care conține rezervările, unde un câmp special, *Status*, indică: dacă o rezervare nu a fost încă începută (*Status Pending*), dacă este în derulare (*Status In progress*) sau dacă a fost finalizată (*Status Done*). Tabelul cu notele este necesar pentru a face posibilă acordarea lor o singură dată, cu orice link, prin introducerea unui cod de verificare în tabel; o notă fiind atribuită doar dacă codul există în tabel și nu a mai fost atribuită o dată.

5.1.1. *Cerințe Funcționale*

Logarea cu două tipuri de conturi este necesară pentru a ascunde funcționalitățile administratorilor de cele ale utilizatorilor normali. Închirierea unei mașini trebuie să fie posibilă doar dacă utilizatorul este logat. Crearea unui cont se poate realiza dacă nu este logat niciun utilizator. Utilizatorii normali pot să vizualizeze lista cu mașinile, în care sunt prezente toate detaliile acestora, și poate să închirieze o mașină, căreia poate să ii acorde o notă. Zilele în care mașina este deja rezervată nu pot fi selectate din calendar. Administratorii pot crea, modifica și șterge mașinile.

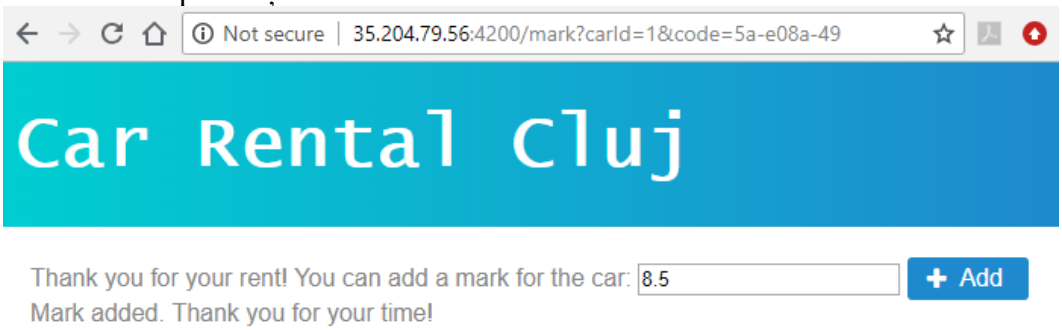
5.1.2. *Cazuri de utilizare principale*

Cazul principal de utilizare este închirierea unei mașini, pentru care este necesar ca utilizatorul să fie logat. După alegerea mașinii dorite, se deschide pagina de închiriere (care se poate vizualiza și în Figura 5.1) prin apăsarea butonului de la baza anunțului. Aici se poate observa un calendar, unde trebuie selectată perioada pentru care se dorește rezervarea. Procesul se finalizează prin apăsarea butonului *Rent*.



Figură 5.1 Pagina de închiriere

O altă funcționalitate importantă este cea prin care se acordă o notă mașinii închiriate. În momentul în care o mașină este preluată, în tabelul cu note se crează o notă fără valoare mașinii respective, având un cod de verificare. După ce a fost returnată mașina, un *link* către pagina în care se acordă notele (Figura 5.2) este primit pe mail, *link* care conține codul de verificare. Acest cod este transmis împreună cu id-ul mașinii și nota atribuită către server, unde nota este salvată doar dacă există, în tabelul notelor, o notă care are codul respectiv și nicio notă nu i-a mai fost atribuită.



Figură 5.2 Pagina notelor

5.1.3. Arhitectură

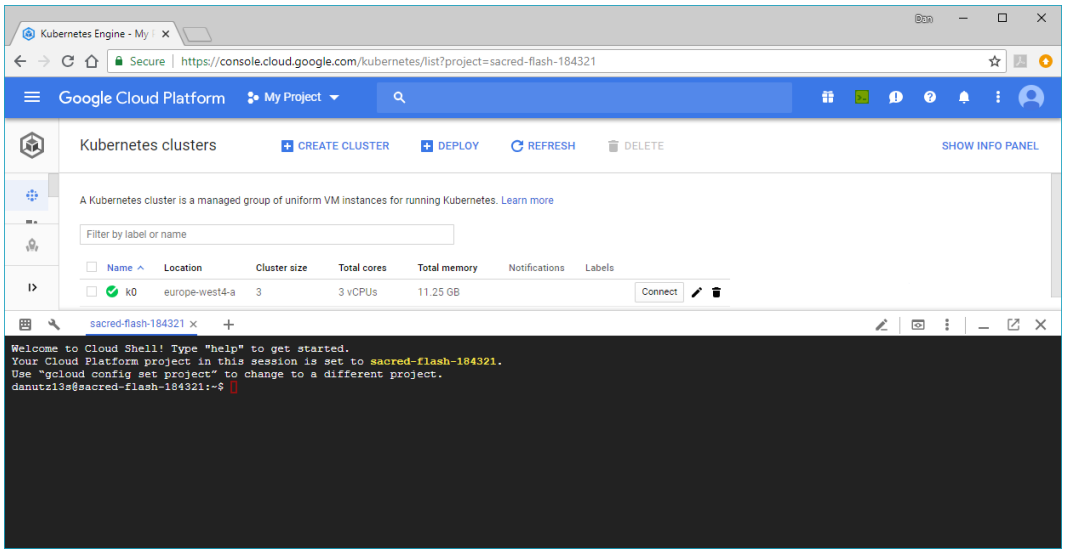
Aplicația folosește o bază de date MySql, la care se conectează un server REST Spring, găzduit în Tomcat. Partea de frontend este scrisă în Angular TypeScript, de unde sunt făcute cererile http către server.

5.2. Google Kubernetes Engine

Kubernetes Engine este un mediu gestionat, pregătit pentru producție, pentru implementarea aplicațiilor containerizate. Acesta aduce cele mai noi inovații în productivitatea dezvoltatorilor, utilizarea eficientă a resurselor, operații automatizate și flexibilitatea open source pentru a accelera timpul de marketing.

Lansat în 2015, Kubernetes Engine se bazează pe experiența acumulată de Google prin rularea serviciilor Gmail și YouTube în containere de peste 12 ani. Kubernetes Engine permite rularea Kubernetes în cel mai scurt timp, eliminând complet necesitatea instalării, gestionării și operării propriilor *Cluster*-e Kubernetes.

Google Cloud Shell este mediul pentru gestionarea resurselor găzduite pe Google Cloud Platform (GCP). Cloud Shell este preinstalat cu instrumentele de linie de comandă *gcloud* și *kubectl*. Gcloud oferă interfața primară a liniei de comandă pentru GCP și *kubectl* furnizează interfața de linie de comandă pentru rularea comenzilor împotriva *Cluster*-elor Kubernetes. Pentru folosirea unui *shell* local este necesară instalarea instrumentelor *gcloud* și *kubectl* pe mașina în pricină. Deschiderea GCShell se face prin apăsarea butonului >_, evidențiat în Figura 5.3.



Figură 5.3 Interfața web a Google Cloud Platform

Un *Cluster* este compus din cel puțin o mașină de tip master și mai multe mașini de lucru numite noduri. Nodurile sunt instanțe ale mașinii virtuale Compute Engine, care execută procesele Kubernetes necesare pentru a face parte din *Cluster*. Aplicațiile lansate în *Cluster* rulează în noduri.

Pentru a crea un *Cluster*, se execută următoarea comandă:

gcloud container clusters create CLUSTER_NAME

Având grupul creat, comenzile *kubectl* pot să fie executate. Pentru crearea spațiului de stocare necesar pentru MySQL, primul pas este crearea PersistentVolumeClaims. Când se creează acesta, dacă nu există niciun PersistentVolume cu care să se lege, un nou PersistentVolume este furnizat în mod dinamic pe baza configurației StorageClass.

Kubernetes Engine are instalat StorageClass implicit, care permite furnizarea dinamică de PersistentVolumes pe discuri persistente. Atunci când StorageClass nu este specificat în PersistentVolumeClaim, se folosește cel implicit al *Cluster*-ului.

Fișierul *mysql-volumeclaim.yaml* care conține descrierea acestuia, este evidențiată în figura de mai jos (Figura 5.4).

```
1  kind: PersistentVolumeClaim
2  apiVersion: v1
3  metadata:
4    name: mysql-volumeclaim
5  spec:
6    accessModes:
7      - ReadWriteOnce
8    resources:
9      requests:
10       storage: 200Gi
```

Figură 5.4 *mysql-volumeclaim.yaml*

PersistentVolumeClaim care cere 200Gi de stocare formatat cu un sistem de fișiere, se aplică cu comanda *kubectl apply -f mysql-volumeclaim* și se verifică cu comanda *kubectl get pvc*.

Pentru lansarea MySQL este necesară crearea unui Kubernetes Secret care stochează parola bazei de date. Pentru a crea secretul *mysql* se folosește comanda *kubectl create secret generic mysql --from-iteral=password=YOUR_PASSWORD*. În Figura 5.5 se poate observa crearea secretului *mysecret* cu o singură valoare, care are numele *my* și valoarea *secret*. Valoarea este codificată în *base64*.

```
danutz13s@sacred-flash-184321:~$ kubectl create secret generic mysecret --from-literal=my=secret
secret "mysecret" created
danutz13s@sacred-flash-184321:~$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  my: c2VjcmV0
kind: Secret
metadata:
  creationTimestamp: 2018-07-06T12:19:53Z
  name: mysecret
  namespace: default
  resourceVersion: "3896118"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: e318418e-8116-11e8-b727-42010aa40085
type: Opaque
danutz13s@sacred-flash-184321:~$ base64 --decode <<<"c2VjcmV0"; echo;
secret
```

Figură 5.5 *Creare și verificare Secret*

În Figura 5.6 este descrisă o singură instanță *Pod* MySQL care va avea variabila de mediu MYSQL_ROOT_PASSWORD a cărei valoare este setată din secretul creat. Containerul *mysql* va folosi PersistentVolumeClaim și va monta discul persistent la */var/lib/mysql* în interiorul containerului. Crearea acestuia se face cu comanda *kubectl create -f mysql.yaml* și se poate verifica cu comanda *kubectl get pod -l app=mysql*.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:5.6
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql
                  key: password
            - name: MYSQL_DATABASE
              value: spring-demo
          ports:
            - containerPort: 3306
              name: mysql
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-volumeclaim
```

Figură 5.6 *mysql.yaml*

Următorul pas este crearea unui serviciu pentru a expune containerul MySQL și pentru a-l face accesibil din containerul serverului. Fișierul *mysql-service.yaml* care conține descrierea acestuia este ilustrat în Figura 5.7.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  type: ClusterIP
  ports:
    - port: 3306
  selector:
    app: mysql
```

Figură 5.7 *mysql-service.yaml*

Acest manifest descrie un serviciu care creează o adresă de *Cluster* pe portul 3306 pentru *Pod*-urile care corespund aplicației cu eticheta: *mysql*. Containerul *mysql* implementat în etapa anterioară are această etichetă. În acest caz, se utilizează tipul: ClusterIP pentru serviciu, deoarece această valoare face ca serviciul să fie accesibil numai din interiorul *Cluster*-ului.

IP-ul *Cluster*-ului a creat traficul către containerul MySQL de la toate nodurile din *Cluster* și nu este accesibil clienților din afara *Cluster*-ului. Odată ce serviciul este creat, containerul serverului poate folosi numele DNS al containerului *mysql* pentru a comunica, chiar dacă acestea nu se găsesc în același nod.

Serviciul se lansează cu comanda *kubectl create -f mysql-service.yaml* și se poate verifica cu comanda *kubectl get svc mysql*. Conectarea la acest server se poate face și prin crearea unui *Pod* care rulează clientul MySQL și se conectează la server prin intermediul serviciului, după cum se poate observa în Figura 5.8.

```
danutz13s@sacred-flash-184321:~$ kubectl run -it --rm --image=mysql:5.6
--restart=Never mysql-client -- mysql -h mysql -ppassword
If you don't see a command prompt, try pressing enter.

mysql> SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;
+--------------------+
| SCHEMA_NAME        |
+--------------------+
| information_schema |
| #mysql50#lost+found |
| mysql              |
| performance_schema |
| spring-demo        |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Figură 5.8 Conectare la serverul MySQL

Imaginea pentru serverul Spring trebuie construită folosind un server Apache Tomcat în care lansăm serverul Tomcat. Pentru a permite gestiunea serverului Tomcat, trebuie modificat fișierul *tomcat-users.xml* astfel încât acesta să conțină un utilizator (admin/admin) și permisiunile necesare (de exemplu: manager-gui pentru a putea accesa pagina de gestiune a aplicațiilor), prezentate în Figura 5.9.

```xml
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="admin-gui" />
  <role rolename="admin-script" />
  <role rolename="manager-gui" />
  <role rolename="manager-status" />
  <role rolename="manager-script" />
  <role rolename="manager-jmx" />
  <user name="admin" password="admin"
        roles="admin-gui,admin-script,manager-gui,
        manager-status,manager-script,manager-jmx"/>
</tomcat-users>
```

Figură 5.9 *tomcat-users.xml*

Pentru lansarea serverului Tomcat este necesar fișierul WAR construit cu programul Maven. Înainte de a fi construit, trebuie configurată conexiunea cu serverul MySQL. Această conexiune este realizată folosind Java Database Connectivity (JDBC), în care IP-ul serverului MySQL este înlocuit cu numele serviciului MySQL, ca în Figura 5.10.

application.properties

```
################### DataSource Configuration #############
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://mysql:3306/spring-demo
jdbc.username=root
jdbc.password=password
init-db=false
################### Hibernate Configuration ##############
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
################### JavaMail Configuration ###############
smtp.host=
smtp.port=
smtp.protocol=
smtp.username=
smtp.password=
support.email=
```

Figură 5.10 Configurarea JDBC

După ce a fost configurată conectivitatea cu serverul MySQL, proiectul poate fi construit. Comanda care face acest lucru este *mvn package*, împachetând aplicația într-un fișier WAR. Acesta este disponibil în folderul *target/*. Trebuie specificat tipul de fișier dorit după împachetare, în fișierul *pom.xml*, ca în figura de mai jos (Figura 5.11).

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    <modelVersion>4.0.0</modelVersion>
    <groupId>spring.demo</groupId>
    <artifactId>spring-demo</artifactId>
    <name>${project.artifactId}</name>
    <version>1.0</version>
    <packaging>war</packaging>

    <properties>
```

Figură 5.11 *pom.xml*

Fișierul Docker necesar construirii imaginii (Figura 5.12) folosește direct codul acestuia, motiv pentru care este extrasă librăria java. Apache Tomcat este downloadat

folosind comada *curl* și apoi este extras codul acestuia. WAR-ul serverului este copiat în directorul *webapps/* și xml-ul cu utilizatorii în directorul *conf/*. Serverul Tomcat este pornit folosind în terminal scriptul *bin/startup.sh*, urmărindu-se log-urile acestuia. În final, portul 8080 este expus pentru a putea fi accesibil din exterior.

```
1  FROM java
2  MAINTAINER me
3
4  RUN curl -O http://archive.apache.org/dist/tomcat/
   tomcat-7/v7.0.55/bin/apache-tomcat-7.0.55.tar.gz
5  RUN tar xzf apache-tomcat-7.0.55.tar.gz
6  ADD spring-demo.war apache-tomcat-7.0.55/webapps/
7  ADD tomcat-users.xml apache-tomcat-7.0.55/conf/
8  CMD apache-tomcat-7.0.55/bin/startup.sh && tail -f
   apache-tomcat-7.0.55/logs/catalina.out
9  EXPOSE 8080
```

Figură 5.12 Tomcat Dockerfile

Având fișierele *spring-demo.war*, *tomcat-users.xml* și Dockerfile-ul de mai sus în același loc, comanda *docker build -t danutz13s/myrepo:v3* crează imaginea serverului Tomcat. Din denumirea acesteia putem să deducem numele de utilizator Docker Hub (danutz13s), registrul în care se salvează imaginea (myrepo) și versiunea acesteia (v3). Contul Docker Hub trebuie creat înaintea construirii imaginii pentru că numele de utilizator este necesar în numele imaginii, iar având un cont, trebuie creat registrul în care se urcă imaginea, folosind aplicația lor web. Pentru a urca imaginea pe hub este necesară logarea cu comanda *docker login*, după care se poate executa comanda propriu-zisă de urcare, *docker push danutz13s/myrepo:v3*.

Având imaginea în Docker Hub, se poate crea containerul. Imaginea se specifică exact în felul în care a fost urcată pe hub, *danutz13s/myrepo:v2*. Deployment-ul serverului Tomcat se poate observa în Figura 5.13.

```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    labels:
5      run: mytom
6    name: mytom
7    namespace: default
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       run: mytom
13   template:
14     metadata:
15       labels:
16         run: mytom
17     spec:
18       containers:
19       - image: danutz13s/myrepo:v2
20         imagePullPolicy: IfNotPresent
21         name: mytom
22         ports:
23         - containerPort: 8080
24           protocol: TCP
```

Figură 5.13 Descrierea YAML a Deployment-ului Tomcat

Serviciul Tomcat poate fi observat în Figura 5.14. Tipul serviciului a fost ales LoadBalancer pentru a fi accesibil din afara grupului. O alternativă ar fi tipul NodePort,

unde trebuie deschis portul respectiv pe Google Cloud Platform, dar varianta cea mai bună ar fi un simplu ClusterIP, pentru a nu avea acces la el din exterior, sporind securitatea.

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: mytom
  name: mytom
  namespace: default
spec:
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 30180
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    run: mytom
  sessionAffinity: None
  type: LoadBalancer
```

Figură 5.14 Descrierea YAML a Service-ului Tomcat

După crearea serverului Tomcat, folosind cele două fișiere YAML de mai sus cu comanda *docker create*, putem să trecem la partea de frontend. Aici trebuie verificate url-urile apelate din serviciile pentru mașini și utiliatori, fiind necesară schimbarea localhost cu IP-ul extern al serviciului Tomcat (Figura 5.15). În Figura 5.16 se poate observa url-ul folosit pentru request-urile mașinilor.

```
danutz13s@sacred-flash-184321:~$ kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE
kubernetes   ClusterIP      10.15.240.1     <none>         443/TCP          32d
myapp        LoadBalancer   10.15.246.194   35.204.79.56   4200:30995/TCP   1d
mysql        ClusterIP      10.15.247.9     <none>         3306/TCP         2d
mytom        LoadBalancer   10.15.246.200   35.204.28.35   8080:31119/TCP   1d
danutz13s@sacred-flash-184321:~$
```

Figură 5.15 Detalii Servicii

```typescript
@Injectable()
export class CarService {
  private carsUrl = 'http://35.204.28.35:8080/spring-demo/car'

  constructor(private http: HttpClient) { }

  async getCars (): Promise<Car[]> {
    return await this.http.get<Car[]>(this.carsUrl + '/all').toPromise();
  }
```

Figură 5.16 URL folosit la request-urile mașinilor

Fiind realizată în Angular TypeScript, construirea imaginii se bazează pe imaginea Node 8 din Docker Hub, preluată cu comanda *FROM node:8*. Se crează directorul */usr/src/app* în care o să fie plasată aplicația prin comanda *RUN mkdir -p /usr/src/app*, după care navigăm spre acesta schimbând directorul de lucru cu comanda *WORKDIR /usr/src/app*. Se copiază fișierul *package.json* în noul director, unde sunt instalate dependențele salvate în el folosind *RUN npm install*. Se copiază și restul fișierelor în directorul aplicației, se expune portul 4200 și se execută *CMD [“npm”, „start”]*. Astfel, fișierul Docker pentru aplicație arată ca în Figura 5.17.

```
FROM node:8

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY package.json /usr/src/app
RUN npm install

COPY . /usr/src/app
EXPOSE 4200

CMD ["npm", "start"]
```

Figură 5.17 Angular Web Application Dockerfile

Exact ca în cazul imaginii Tomcat, trebuie creat un registru în Docker Hub, după care se poate urca imaginea în el. Construirea imaginii se face cu comanda *docker build -t danutz13s/myapp:v3* și urcarea sa în hub cu *docker push danutz13s/myapp:v3*.

Având imaginea pregătită, fișierul YAML care conține detaliile Deployment-ului poate fi creat. Se specifică tipul (Deployment), numele (myapp), eticheta (myapp), imaginea (danutz13s/myapp:v3) și portul (4200), după cum se poate observa în Figura 5.18.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: myapp
  name: myapp
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: myapp
  template:
    metadata:
      labels:
        run: myapp
    spec:
      containers:
      - image: danutz13s/myapp:v3
        imagePullPolicy: IfNotPresent
        name: myapp
        ports:
        - containerPort: 4200
          protocol: TCP
```

Figură 5.18 Descrierea YAML a Deployment-ului aplicației

Serviciul care expune *Pod*-ul aplicației trebuie să conțină numele (myapp), eticheta (myapp), porturile (4200) și tipul (LoadBalancer) serviciului, după cum se poate vizualiza și în Figura 5.19.

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: myapp
  name: myapp
  namespace: default
spec:
  clusterIP: 10.15.246.194
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 30995
    port: 4200
    protocol: TCP
    targetPort: 4200
  selector:
    run: myapp
  sessionAffinity: None
  type: LoadBalancer
```

Figură 5.19 Descrierea YAML a Service-ului aplicației

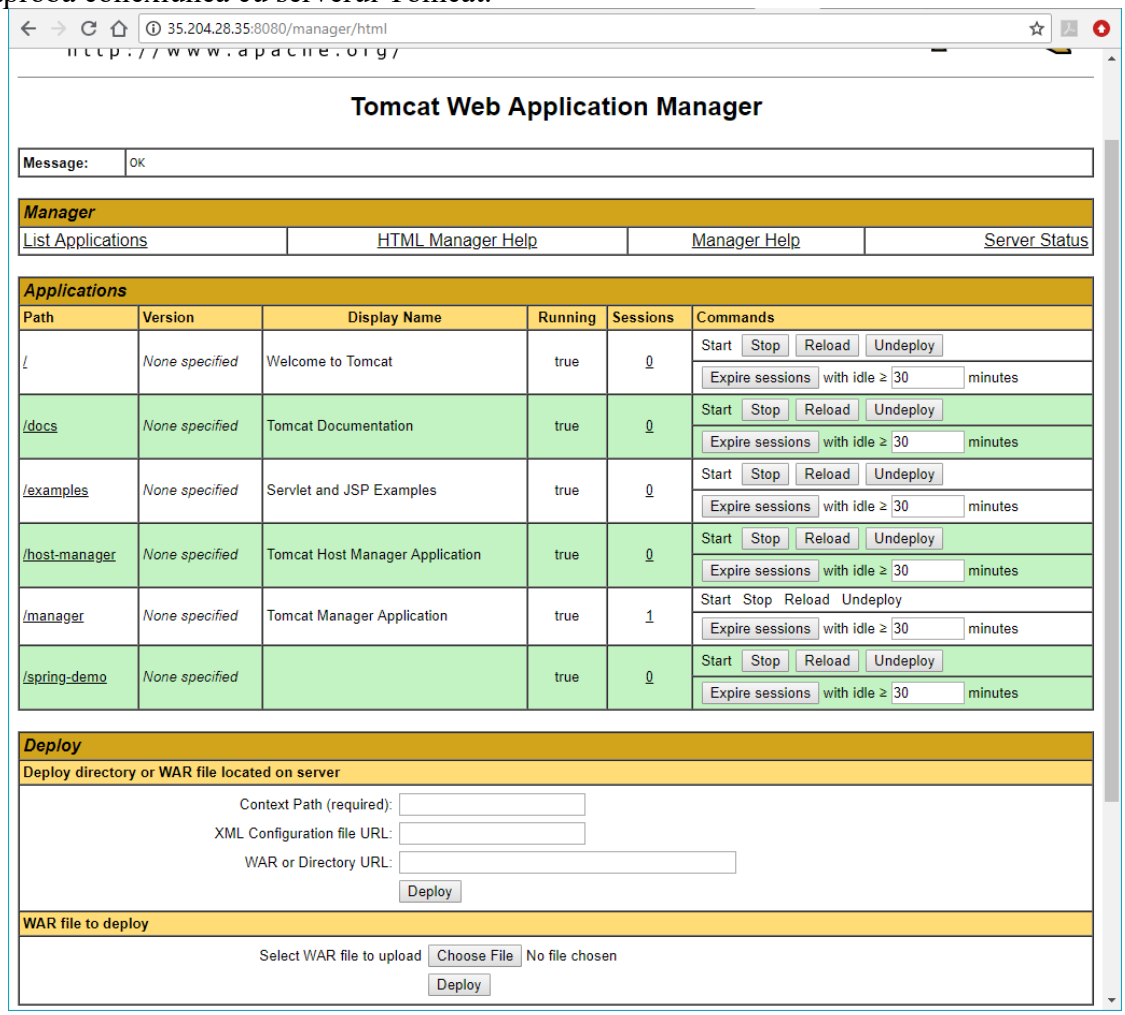
Astfel, imaginile au fost construite cu comanda *docker build* și urcate în Docker Hub cu *docker push*. Componentele descrise mai sus au fost create cu *kubectl create*, asftel având stocarea persistentă pentru serverul MySQL, Deployment-ul și Serviciul pentru MySQL, Tomcat și aplicația Web Angular. Legăturile au fost făcute, deci aplicația trebuie să funcționeze. Verificarea acestor componente se face cu comanda *kubectl get*, urmată de *pv* pentru *Persistent Volume*, *pvc* pentru *Persistent Volume Claim*, *deploy* pentru *Deployment* și *svc* pentru *Service* (Figura 5.20).

```
danutz13s@sacred-flash-184321:~$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
pvc-cd5e1f82-7e97-11e8-ac86-42010aa40085   2Gi        RWO            Delete           Bound
danutz13s@sacred-flash-184321:~$ kubectl get pvc
NAME                STATUS    VOLUME                                     CAPACITY   ACCESS MOD
mysql-volumeclaim   Bound     pvc-cd5e1f82-7e97-11e8-ac86-42010aa40085   2Gi        RWO
danutz13s@sacred-flash-184321:~$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp     1         1         1            1           1d
mysql     1         1         1            1           2d
mytom     1         1         1            1           1d
danutz13s@sacred-flash-184321:~$ kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE
kubernetes   ClusterIP      10.15.240.1     <none>         443/TCP          32d
myapp        LoadBalancer   10.15.246.194   35.204.79.56   4200:30995/TCP   1d
mysql        ClusterIP      10.15.247.9     <none>         3306/TCP         2d
mytom        LoadBalancer   10.15.246.200   35.204.28.35   8080:31119/TCP   1d
danutz13s@sacred-flash-184321:~$
```

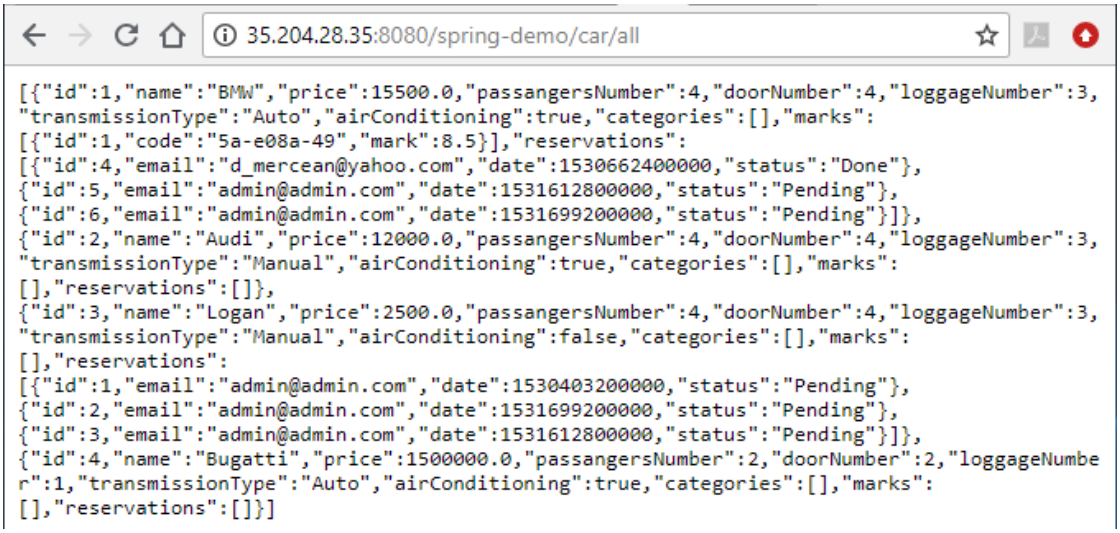
Figură 5.20 Componentele aplicației

În figura de mai sus putem observa IP-urile externe ale serviciului Tomcat (35.204.28.35) si ale serviciului aplicației Angular (35.204.79.56). Accesând aceste IP-uri trebuie sa fim capabili să folosim serviciile respective. Asftel, accesând IP-ul serverului tomcat trebuie să apară pagina acestora de pornire, de unde putem accesa pagina de gestiune apăsând pe butonul Manager App (Figura 5.21). Aici o să fie cerută logarea utilizatorului, unde se folosesc credențialele create în fișierul *tomcat-users.xml*, numele de utilizator și parola din acesta fiind *admin*. Se pot verifica endpoint-urile serverului Spring găzduit în acesta prin navigarea la */spring-demo/car/all* (Figura 5.22). Mașinile cu toate detaliile lor sunt primite în formatul JSON, deci conexiunea la baza de date funcționează. Aplicația poate fi accesată folosind IP-ul extern al serviciului său (Figura

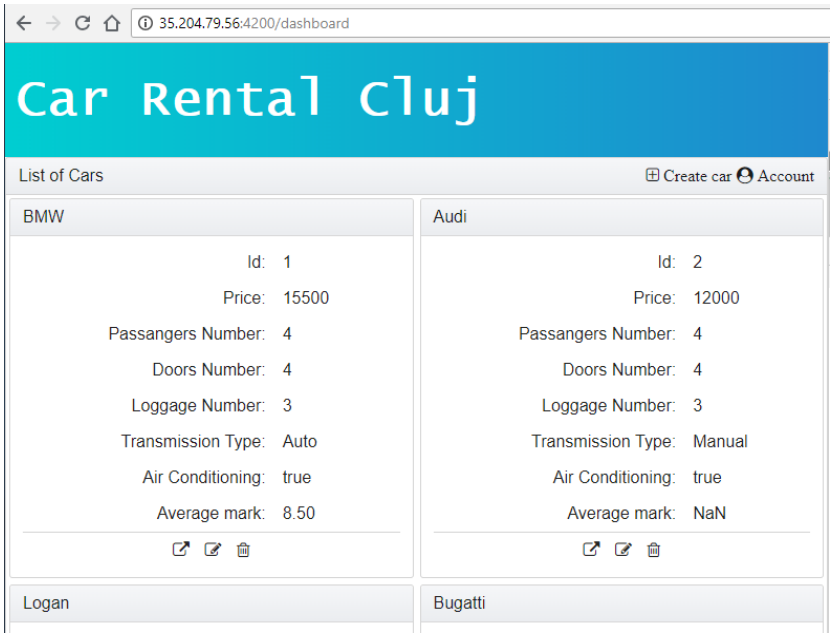
5.23), unde ar trebui să apară pagina principară și să apară mașinile de închiriat, fapt ce aprobă conexiunea cu serverul Tomcat.



Figură 5.21 Tomcat Web Application Manager



Figură 5.22 Request-ul pentru a primi toate mașinile



Figură 5.23 Aplicația de frontend

Scalarea aplicației se face cu comanda *kubectl scale deployment myapp --replicas 2*, care crează un nou Pod pe lângă cel existent deja. După execuție, se pot observa ambele Pod-uri cu comanda *kubectl get deployment* (Figura 5.24).

```
danutz13s@sacred-flash-184321:~$ kubectl scale deployment myapp --replicas 2
deployment "myapp" scaled
danutz13s@sacred-flash-184321:~$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp     2         2         2            2           1d
mysql     1         1         1            1           2d
mytom     1         1         1            1           2d
danutz13s@sacred-flash-184321:~$
```

Figură 5.24 Scalarea Deployment-ului aplicației

Această scalare se poate efectua automat folosind *Horizontal Pod Autoscaler-ul* (HPA), care crează sau șterge *Pod*-uri după pragul de utilizare al procesorului. Astfel, în Figura 5.25 este descris HPA-ul care scalează automat *Deployment*-ul myapp dacă procentul de folosire al procesorului depășește 80, având numărul minim de replici 3 și maxim 6. Crearea acestuia se face cu comanda *kubectl create* și se verifică cu *kubectl get hpa* (Figura 5.26).

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-hpa
spec:
  minReplicas: 3
  maxReplicas: 6
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: myapp
  targetCPUUtilizationPercentage: 80
```

Figură 5.25 HPA pentru autoscalarea Deployment-ului aplicației

```
danutz13s@sacred-flash-184321:~$ kubectl create -f myapp-hpa.yaml
horizontalpodautoscaler "myapp-hpa" created
danutz13s@sacred-flash-184321:~$ kubectl get hpa
NAME        REFERENCE          TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
myapp-hpa   Deployment/myapp   0% / 80%   3         6         3          5m
danutz13s@sacred-flash-184321:~$
```

Figură 5.26 Crearea și verificarea HPA-ului

Deployment-urile permit actualizarea prin comanda *kubectl set*, care actualizează imagininea Deployment-lui. Actualizarea imaginii Deployment-ului myapp la danutz13s/myapp:v2 se face cu *kubectl set image deployment/myapp myapp= danutz13s/myapp:v2* (Figura 5.27). Verificarea acestei comenzi se poate face cu *kubectl rollout status deployment/myapp*, care ar trebui să returneze *deployment "myapp" successfully rolled out*.

```
danutz13s@sacred-flash-184321:~$ kubectl set image deployment/myapp myapp=danutz13s/myapp:v2
deployment "myapp" image updated
danutz13s@sacred-flash-184321:~$ kubectl rollout status deployment/myapp
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "myapp" successfully rolled out
danutz13s@sacred-flash-184321:~$
```

Figură 5.27 Setarea imaginii Deployment-ului aplicației și verificarea acestuia

Dacă schimbarea unei imagini a dus la eșuarea Pod-urilor, comanda *kubectl rollout undo deployment/myapp* face *rollback* la versiunea anterioară. Schimbarea de imagine din Figura 5.27 a trecut de la versiunea 3 la versiunea 2 a imaginii danutz13s/myapp, astfel încât, un rollout undo trebuie să readucă imaginea la versiunea 3, caz ilustrat în figura de mai jos (Figura 5.28). Imaginea *Deployment*-ului a fost obținută cu comanda *kubectl describe deploy myapp | grep Image*.

```
danutz13s@sacred-flash-184321:~$ kubectl describe deploy myapp | grep Image
    Image:        danutz13s/myapp:v2
danutz13s@sacred-flash-184321:~$ kubectl rollout undo deployment/myapp
deployment "myapp"
danutz13s@sacred-flash-184321:~$ kubectl describe deploy myapp | grep Image
    Image:        danutz13s/myapp:v3
danutz13s@sacred-flash-184321:~$ kubectl rollout status deployment/myapp
deployment "myapp" successfully rolled out
danutz13s@sacred-flash-184321:~$
```

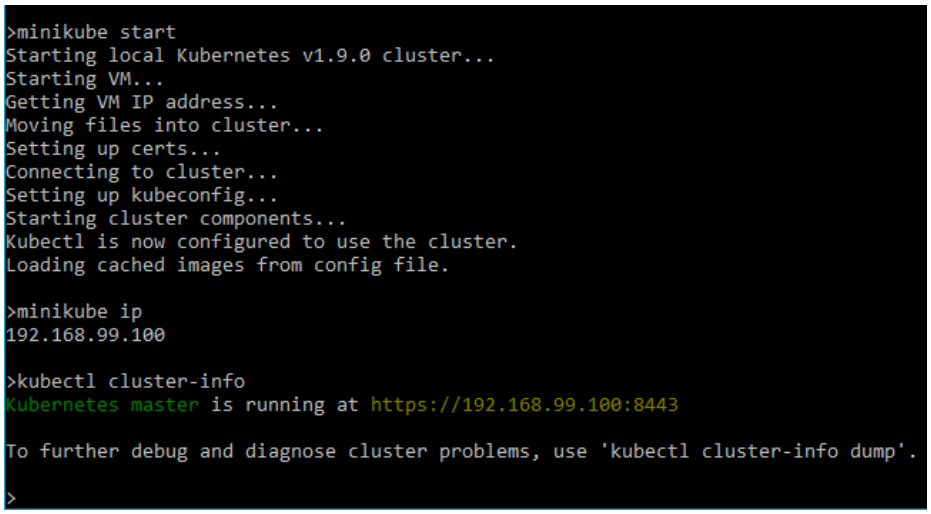
Figură 5.28 Deployment *rollout undo*

5.3. Minikube

Minikube este un instrument care face ușoră rularea Kubernetes pe mașina locală. Minikube rulează un *Cluster* Kubernetes cu un singur nod în interiorul unei mașini virtuale (VM) pentru utilizatorii care doresc să încerce Kubernetes sau să dezvolte cu el zilnic.

Înainte de crearea clusterului Minikube, este nevoie de un hypervisor, precum Oracle VirtualBox, VMware Fusion, Kernel-based Virtual Machine, Kernel-based Virtual Machine 2, HyperKit sau Xhyve. La crearea *Cluster*-ului Minikube cu un singur nod (Figura 5.29), sistemul gazdă trebuie să aibă resursele necesare pentru a-l executa. În mod implicit, Minikube utilizează 1048 MB de memorie RAM și o unitate centrală de procesare. Această configurație poate fi modificată cu comanda *minikube start --cpus=2 --memory=2048*. Dacă este folosit un hypervisor diferit de Oracle VirtualBox, trebuie să fie specificat adăugând *--vm-driver=kvm2* comenzii *docker start*.

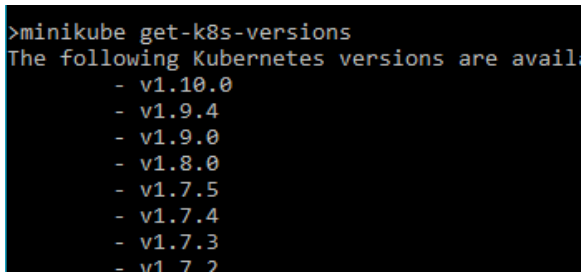
Având *Cluster*-ul cu un singur nod în desfășurare, trebuie să i se afle adresa IP. Aceasta poate fi verificată folosind *minikube ip* (Figura 5.29) sau, dacă sunt necesare informații mai detaliate se poate folosi *kubectl cluster-info* (Figura 5.29). Comanda *cluster-info* arată toate componentele care rulează într-un *Cluster*. Pentru Minikube, există numai Kubernetes master.



Figură 5.29 Pornirea Minikube

Atunci când se utilizează o singură mașină virtuală Kubernetes, este foarte utilă utilizarea daemon-ului Docker încorporat de Minikube deoarece astfel nu trebuie construit un registru Docker pe mașina gazdă și să se urce imaginea în el. Imaginile se pot construi în interiorul aceluiași daemon Docker ca minikube, astfel accelerând dezvoltarea locală. Pentru a putea lucra cu *daemon*-ul Docker de pe gazdă se folosește comanda *eval $(minikube docker-env)*. Acum ar trebui să se poată utiliza Docker în linia de comandă din gardă, vorbind cu *daemon*-ul Docker din interiorul mașinii virtuale Minikube.

Minikube acceptă rularea a diferite versiuni Kubernetes. Lista tuturor versiunilor disponibile se poate accesa cu *minikube get-k8s-versions* (Figura 5.30). Versiunea Kubernetes pe care să o folosească Minikube se poate specifica prin adăugarea *--kubernetes-version* la comanda de pornire Minikube. Rularea versiunii v1.7.3 se face cu *minikube start --kubernetes-version v1.7.3*. În Figura 5.31 se poate observa că, dacă a fost folosită versiunea v1.9.0 înainte, pornirea cu versiunea v1.7.3 (versiune mai mică) nu este posibilă, versiunea rămânând cea mai mare.



Figură 5.30 Versiuni Kubernetes disponibile în Minikube

```
>minikube start --kubernetes-version v1.7.3
Starting local Kubernetes v1.7.3 cluster...
Starting VM...
Getting VM IP address...
Kubernetes version downgrade is not supported. Using version: v1.9.0
Moving files into cluster...
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

Figură 5.31 Pornire Minikube cu versiunea Kubernetes specificată

Minikube acceptă Persistent Volume de tipul *hostPath*. Aceste volume persistente sunt mapate într-un director din interiorul mașinii virtuale Minikube. Minikube VM bootează într-un *tmpfs* (temporary file storage), astfel încât cele mai multe directoare nu vor fi persistente de-a lungul restartărilor (*minikube stop*). Cu toate acestea, Minikube este configurat să persiste fișierele stocate în directoare gazdă */data*, */var/lib/localkube* și */var/lib/docker*.

În Figura 5.32 este descris un Persistent Volume cu permisiuni *ReadWriteOnce*, stocare de 5Gi și path */data/pv0001/*, care este persistent chiar și după oprirea Minikube cu comanda *minikube stop*. Pentru a verifica conținutul acestor directoare se poate folosi *minikube ssh* (Figura 5.33), care permite executarea comenzilor în interiorul *Cluster*-ului *single node*. Numele de utilizator implicit este *docker*, iar parola *tcuser*, necesare în momentul în care se face ssh (*Secure Shell*) din alt program, folosind adresa IP a mașinii virtuale Minikube.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 5Gi
  hostPath:
    path: /data/pv0001/
```

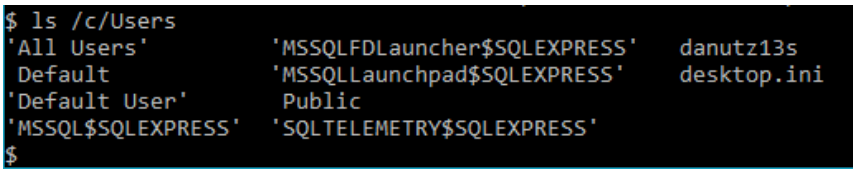
Figură 5.32 Persistent Volume cu path persistent pentru Minikube

```
>minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$ ls /
bin  data  etc   init  lib64   media  opt   root  sbin  sys  usr
c    dev   home  lib   linuxrc  mnt    proc  run   srv   tmp  var
$
```

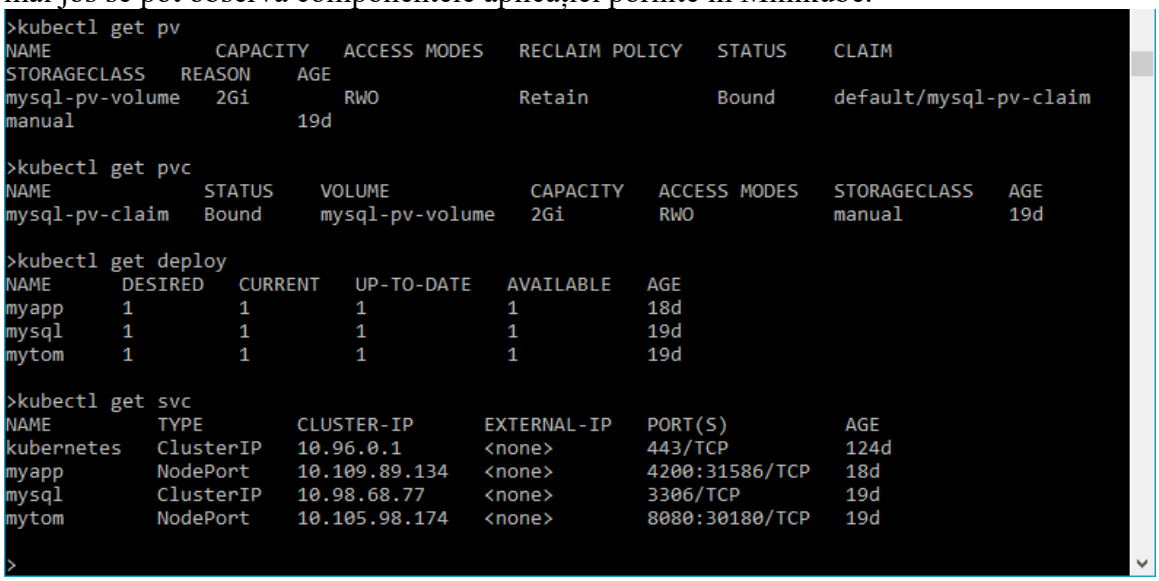
Figură 5.33 Comanda *minikube ssh*

Unele drivere vor monta un folder gazdă în VM, astfel încât să se poată partaja fișiere între VM și gazdă. Acestea nu sunt configurabile și sunt diferite pentru driverul și sistemul de operare utilizat. Folosind VirtualBox, pe sistemele Windows folder-ul *C://Users* este încărcat în */c/Users* (Figura 5.34), în timp ce pe sistemele Linux, folder-ul */home* este încărcat în */hosthome*.



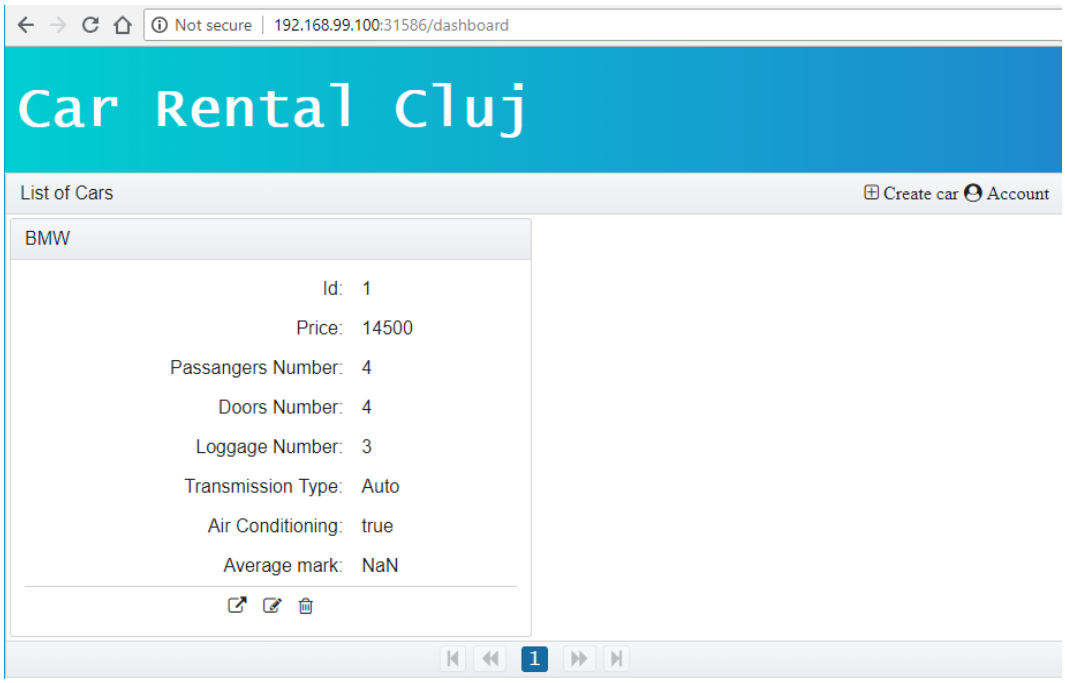
Figură 5.34 Conținutul directorului /c/Users din Minikube

Datorită infrastructurii Kubernetes și din cauza componentelor sale, având Cluster-ul cu un singur nod creat, orice aplicație containerizată descrisă cu fișiere YAML poate să fie lansată. Astfel s-au folosit fișierele de descriere utilizate în Google Kubernetes Engine pentru a porni aplicația local, în Minikube. Datorită faptului că Minikube rulează local, serviciul necesar pentru a expune aplicația este de tipul NodePort, făcând serviciile disponibile pe un port de la adresa IP a nodului. În figura de mai jos se pot observa componentele aplicației pornite în Minikube.



Figură 5.35 Componentele aplicației pornite în Cluster-ul Kubernetes Minikube

În Figura 5.36 se poate observa aplicația de închiriat mașini care rulează pe adresa IP a Cluster-ului Minikube, pe portul 31586. Conexiunea cu serverul Tomcat este validată de mașina care apare în listă și de butonul *Create car*, care este vizibil doar pentru administratori, astfel fiind clar că un utilizator este logat. Serverul Tomcat poate fi accesat similar, folosind portul 30180 de pe adresa IP a nodului Minikube. Astfel, înafară de schimbarea tipului de serviciu de la Load Balancer la Node port, etapele și fișierele folosite sunt aceleași pe orice mediu de lansare a aplicațiilor containerizate cu Kubernetes.



Figură 5.36 Aplicația Angular expusă din Minikube pe portul 31586

Capitolul 6. Testare şi Validare

Pentru a testa virtualizarea cu containere oferită de Docker și gestionată de Kubernetes, s-a folosit un server Apache Tomcat pentru a măsura durata de timp necesară la crearea acestuia. S-au făcut măsurători local cu Minikube și în cloud cu Google Kubernetes Engine. Pentru acest lucru, am descris serverul cu un Deployment și un serviciu (Figura 6.1), care l-am folosit pentru crearea lui cu comanda *kubectl create -f tomcat.yaml*. Deployment-ul este simplu și folosește imaginea tomcat. Serviciul este unul de tipul NodePort pentru a putea fi accesat din afara Cluster-ului și are specificat *clusterIP*-ul, necesar pentru testarea din Google Kubernetes Engine.

```yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Deployment
  metadata:
    labels:
      run: tomcat
    name: tomcat
  spec:
    replicas: 1
    selector:
      matchLabels:
        run: tomcat
    template:
      metadata:
        labels:
          run: tomcat
      spec:
        containers:
        - image: tomcat
          imagePullPolicy: Always
          name: tomcat
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      run: tomcat
    name: tomcat
  spec:
    clusterIP: 10.101.101.227
    externalTrafficPolicy: Cluster
    ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
    selector:
      run: tomcat
    type: NodePort
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

Figură 6.1 *tomcat.yaml*

Pentru măsurarea timpului s-a creat un program Python (Figura 6.2). În acesta sunt efectuate cereri http către serverul Tomcat până în momentul în care vreo cerere returnează statusul *200 OK*. Durata de timp este măsurată cu librăria *time* și metoda *time()*, care returnează numărul de secunde de la *epoch*, un punct de referință de la care se măsoară timpul, folosit pentru a înlătura ambiguitatea evenimentelor care utilizează data sau timpul. Valoarea măsurată este printată pe ecran și salvată într-un fișier. Cererile http pentru scenariul local, care folosește un Cluster Kubernetes Minikube cu un singur nod, sunt făcute la IP-ul nodului, pe portul expus de PortNode.

```
1  import http.client
2  import time
3
4  start = time.time()
5  time.process_time()
6  while True:
7      try:
8          connection=http.client.HTTPConnection("192.168.99.100:30038")
9          connection.request("HEAD","/")
10         status=connection.getresponse()
11         statuscode=status.status
12         if statuscode==200:
13             break
14     except:
15         pass
16
17 time = time.time() - start
18 file = open("C:\\Users\\danutz13s\\Desktop\\output.txt",'w')
19 file.write(str(time))
20 file.close()
```

Figură 6.2 *tomcat.py* pentru Python 3

Măsurarea timpului pentru scenariul din Google Kubernetes Engine este puțin diferită (Figura 6.3). Versiunea Python instalat pe nodurile create în cloud-ul lor este Python 2, așa că au fost necesare câteva ajustări, schimbarea librăriei pentru cererile http și modificarea folosirii librăriei *time*. Intervalul din care este atribuit portul serviciului este 30000-32767, astfel fiind necesară deschiderea lui din Google Cloud Platform. Pentru a evita acest lucru, programul Python a fost rulat pe un nod din cloud, de unde s-au făcut cererile http către IP-ul de Cluster al serviciului, vizibil doar în interiorul lui.

```
import httplib
import time

start = time.time()
while True:
        try:
                connection=httplib.HTTPConnection("10.15.240.227:8080")
                connection.request("HEAD","/")
                status=connection.getresponse()
                statuscode=status.status
                if statuscode==200:
                        break
        except:
                pass

time = time.time() - start
print(time)
file = open("output.txt",'w')
file.write(str(time))
file.close()
```

Figură 6.3 *tomcat.py* pentru Python 2

Pentru a realiza o testare cât de corectă, comenzile de creare a Deployment-ului și a serviciului și cea de execuție a programului de măsurare trebuie să fie executate în același timp. Pentru testul local s-a folosit un fișier *batch* (Figura 6.4) din care s-au executat comenzile cu *start*, care apelează comenzile în terminale noi.

```
1  start kubectl apply -f tomcat.yaml
2  start python tomcat.py
```

Figură 6.4 *tomcat.bat*

Pentru lansarea simultană a comenzilor în Google Kubernetes Engine s-a create un fișier *bash* (Figura 6.5) în care s-au apelat cele două comenzi. Rularea fișierului Python pe Master nu este posibilă fără deschiderea portului de pe nod, așa că s-a folosit o conexiune *ssh* pentru a apela fișierul din interior, de unde se pot face request-uri http către adresa IP internă a serviciului. Conexiunea cu nodul de pe care se execute fișierul Python s-a facut cu *gloud compute ssh*, în care, comanda a fost specificată cu parametrul *–command=”python tomcat.py”*. La apelare, se salvează identificatorii celor două procese pentru a aștepta terminarea acestora cu comanda *wait*. Comanda *trap* oprește subprocesele dacă procesul principal este închis.

```
#!/bin/bash

kubectl apply -f tomcat.yaml & pid=$!;
PID_LIST+=" $pid";
gcloud compute ssh gke-k0-default-pool-1935ba2c-j1kk --zone=europe-west4-a --command="python tomcat.py" & pid=$!;
PID_LIST+=" $pid";

trap "kill $PID_LIST" SIGINT

echo "Parallel processes have started";

wait $PID_LIST

echo
echo "All processes have completed";
```

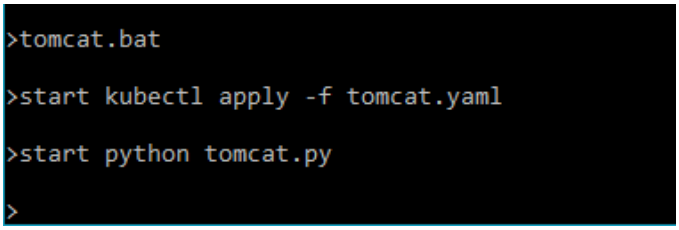
Figură 6.5 *parallel_commands.sh*

După execuția comenzilor se poate observa cât timp a fost necesar pâna când serviciul a fost capabil să răspundă solicitărilor. După rularea de mai mult ori s-a văzut că durata din Google Kubernetes Engine poate varia, astfel având momente în care este mai rapid decât Minikube. Pe de altă parte, Minikube este mai constant, având o performanți asemănătoare mereu. Execuția locală se face cu asd (Figura 6.6) și cea din Google Kubernetes Engine cu ./parallel_commands.sh(Figura 6.7). Afișarea timpului nu se poate vedea când este folosit fișierul Batch pentru că comenzile sunt executate in Command Prompt-uri (*cmd*) diferite, valoarea fiind printată în alt cmd. Aceasta este vizibilă în fișierul output.txt și are valoarea *7.431463718414307*.

```
danutz13s@sacred-flash-184321:~$ ./parallel_commands.sh
Parallel processes have started
deployment "tomcat" created
service "tomcat" created
5.59976792336

All processes have completed
```

Figură 6.6 Execuția pe Google Kubernetes Engine



Figură 6.7 Execuția locală pe Cluster-ul Kubernetes cu un singur nod, Minikube

Capitolul 7. Concluzii

Am aprofundat virtualizarea cu containere și componentele sale care permit o izolare similară cu mașinile virtuale create cu Hypervisors, dar care au o performanță mai bună. Hypervisor-ii permit rularea unui sistem de operare oaspete diferit de sistemul de operare gazdă, dar acest lucru îi face să ocupe mai mult spațiu pe disc și se întreține mai greu. Am observat că, containerele necesită doar aplicația și dependențele sale și că nucleul este împărțit între ele. Pentru că sistemul de operare rulează deja, pornirea unui container este mult mai rapidă decat pornirea unei mașini virtuale.

Am explorat principalele funcționalități Docker, cum ar fi construirea unei imagini, rularea unei imagini, urcarea sau descărcarea unei imagini în Docker Hub sau vizionarea log-urilor containerelor. Am analizat sistemele speciale Docker. Docker Swarm, un instrument de grupare, care ia mai multe motoare Docker și le expune ca o singură instanță. Docker Machine, un instrument pentru gestiunea containerelor care oferă toate comenzile de bază necesare acestui lucru. Docker Compose, un instrument care are capacitatea de a conecta mai multe containere, permițând să se compună o aplicație complexă din mai multe imagini.

Din arhitectura generală Kubernetes am observat conceptele care stau la baza acestuia și care duc la construirea stivelor de servicii și aplicații. Am înțeles mai bine modul în care aceste abstracții ușurează gestionarea ciclului de viață al stivei și al serviciilor în ansamblul lor și nu doar al componentelor individuale. În plus, am analizat cum să gestionăm anumite sarcini de zi cu zi, folosind Pods, Services și Replication Controllers. Am explorat noua abstracție de implementare (Deployment) și modul în care aceasta îmbunătățește Replication Controller-ul.

Am observat cum se aplică manual scalarea în Kubernetes. De asemenea, am analizat funcțiile încorporate pentru a rula actualizările, precum și integrarea lentă a actualizărilor. Am aruncat o privire asupra modului de extindere a nodurilor Cluster-ului. Am explorat unele concepte de scalare automată pentru aplicații. Am văzut beneficiile aduse de Deployment, care permite actualizarea ușoară a imaginilor Pod-urilor sale și integrarea solidă cu serviciile și autoscalarea.

Am explorat o varietate de opțiuni de stocare persistente și cum pot fi implementate împreună cu Pod-urile. Analizând PersistentVolumes și, de asemenea, PersistentVolumeClaims, am observat că acestea permit separarea furnizorului de stocare și a cererilor de stocare a aplicațiilor. În plus, am observat cum StorageClasses specifică furnizorul de stocare conform unei specificații.

Am folosit o aplicație alcătuită dintr-un server Spring, găzduit într-un server Apache Tomcat, care se leagă la o bază de date MySQL. Serverul este folosit de o aplicație Web Angular, care are ca scop închirierea de mașini. Astfel, am folosit Persistent Volume și Persistent Volume Claim pentru a oferi stocare persistentă serverului MySQL. Am folosit Deployment-uri și Service-uri pentru a crea cele 3 instanțe, MySQL, server Tomcat și aplicație Angular. Am observat tipurile de servicii, care pot fi vizibile doar în Cluster (ClusterIP), care se expun pe un port al nod-ului (NodePort) sau care sunt expuse printr-un IP extern (LoadBalancer). Am observat că acestea sunt, de fapt, extensii, deoarece un LoadBalancer folosește un NodePort și un ClusterIP și un NodePort folosește un ClusterIP. Astfel, fiecare fiind o extensie a celui de

înaintea lui. Am observat că aceleași fișiere de configurare și imagini permit pornirea aplicației pe mai multe platforme din cauza infrastructurii Kubernetes.

Am observat că Minikube este un instrument care face ușoră rularea Kubernetes pe mașina locală, rulând un *Cluster* Kubernetes cu un singur nod în interiorul unei mașini virtuale (VM) pentru utilizatorii care doresc să încerce Kubernetes sau să dezvolte cu el zilnic. Am văzut cum accelerează Minikube dezvoltarea locală deoarece acesta, fiind un Cluster cu un singur nod, permite utilizarea daemon-ului Docker încorporat, astfel fiind capabilă construirea imaginilor în interiorul aceluiași daemon Docker ca minikube. Acestu lucru reduce timpul pierdut pentru a urca și descărca imaginile dintr-un registru Docker.

Am realizat un test de viteză pentru crearea unui server Tomcat. Am lansat comanda de pornire a serverului și am realizat request-uri http până în momentul în care acesta este funcțional. După rularea de mai mult ori s-a văzut că durata din Google Kubernetes Engine poate varia, astfel având momente în care este mai rapid decât Minikube. Pe de altă parte, Minikube este mai constant, având o performanți asemănătoare mereu.

Din acest studiu am putut observa că gestiunea mașinilor virtuale poate deveni o sarcină dificilă, deoarece există multe instrumente de virtualizare, astfel încât o companie este posibil să dețină mai multe mașini virtuale sau containere create cu diferiți furnizori. Kubernetes este *open source*, oferind libertatea de a profita de infrastructura cloud locală, hibridă sau publică, făcând posibilă mutarea fără efort a volumului de lucru oriunde este necesar.

Bibliografie

[1] Vladimír Jurenka, "Virtualization using Docker Platform", Master's thesis 2015.
[2] Gigi Sayfan, „Mastering Kubernetes - Automating container deployment and management", Packt Publishing, 2017.
[3] Jonathan Baier, „Getting Started with Kubernetes - Harness the power of Kubernetes to manage Docker deployments with ease", Packt Publishing, Second Edition, 2017.
[4] Miika Moilanen, „Deploying an application using Docker and Kubernetes", Bachelor's thesis, 2018.
[5] Ondrej Sejvl, „Suitability analysis of Kubernetes for Seznam.cz", Master's thesis, 2016.
[6] Documentație Google Kubernetes Engine, https://cloud.google.com/kubernetes-engine/docs/.
[7] Documentație Kubernetes, https://kubernetes.io/docs/home/.

Anexa 1 - Lista figurilor din lucrare

Anexa 2 – Glosar de termeni

KVM	Kernel-based Virtual Machine
YAML	Ain't Markup Language
UTS	Unix timestamp sharing
IPC	Interprocess communication
POSIX	Portable Operating System Interface
PID	Process ID
lo	loopback
CPU	Central processing unit
LXC	Linux Containers
API	Application Programming Interface
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SSH	Secure Shell
IP	Internet Protocol
GKE	Google Kubernetes Engine
RC	Replication Controller
HPA	Horizontal Pod Autoscaler
PV	Persistent Volumes
PVC	Persistent Volume Claims
SC	StorageClass
GCP	Google Cloud Platform
CMD	Command Prompt