UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Luka Jovičić

INFRASTRUKTURA KAO KOD U OKRUŽENJIMA JAVNOG OBLAKA

master rad

Beograd, 2024.

Mentor:

dr Aleksandar KARTELJ, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Vladimir FILIPOVIĆ, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Saša MALKOV, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Infrastruktura kao kod u okruženjima javnog oblaka

Rezime: U ovom radu, predstavićemo jednu metodu postavljanja i podešavanja infrastrukture u javnom oblaku. U prvom delu rada ćemo ukratko opisati ključne osobine računarstva u oblaku i Kubernetes sistema za automatizovanje i isporučivanje aplikacija u kontejnerima, a zatim ćemo opisati alat za opisivanje infrastrukture kodom Teraform i pružiti primer jedne jednostavnije infrastrukture u Teraformu. Nakon toga, prikazaćemo kako pomoću alata Pulumi možemo opisivati infrastrukturu u oblaku koristeći konvencionalne programske jezike. Sastavni deo ovog rada je projekat, dostupan na `https://github.com/luka-j/pulumi-oci-k8s`, koji opisuje infrastrukturu na Oraklovoj infrastrukturi u oblaku, podešava Kubernetes i alate neophodne za neprekidnu isporuku proizvoljnih aplikacija, čini ih javno dostupnim na zasebnim poddomenima i postavlja GitHub repozitorijume. Ovaj projekat se može koristiti za postavku infrastrukture za potrebe organizacije koja broji i desetine timova programera koji su odgovorni za razvoj po jedne ili više aplikacija. Naš cilj je pokazati da je pomoću alata za opisivanje infrastrukture kodom moguće postaviti relativno kompleksna okruženja bez velikih ulaganja, potpuno automatizovano, na način koji je i vremenski i finansijski isplativ i da pružimo demonstraciju ovog koncepta kroz program koji je podesiv, ali sa podrazumevanim podešavanjima upotrebljiv uz minimalnu prethodnu postavku.

Ključne reči: infrastruktura kao kod, oblak, Teraform, Pulumi, Kubernetes

Sadržaj

Glava 1

Okruženja u oblaku

Sve više softvera koji svakodnevno koristimo se u manjoj ili većoj meri oslanja na pristup internetu i preuzimanje podataka ili izvršavanje logike na udaljenim računarima. Uspostavljanje takvih sistema je u ranim danima interneta zahtevalo nabavku, podešavanje i upravljanje računarima i pratećim hardverom. Ovo je iziskivalo ogromna ulaganja i nije bilo preterano praktično ni za koga, tako da se već 1994. pojavljuju servisi poput *GeoCities*-a [14] koji omogućavaju drugim ljudima da postave nešto na internet bez dodira sa konkretnim hardverom. Prvi takvi servisi su bili daleko od zadovoljavajućih za firme koje su želele da u svoje proizvode ugrade funkcionalnosti koje bi zahtevale udaljene računare, ali predstavljaju začetak jedne nove industrije.

Koncept je vremenom evoluirao, tako da danas postoji veliki broj platformi koje omogućavaju pojedincima i firmama da iznajme virtuelne mašine, diskove ili IP adrese, ili čak da pokreću kontejnere, ne razmišljajući o tome šta se dešava u pozadini. Među ovim platformama postoje ogromne razlike, od načina funkcionisanja i cenovnika, do funkcionalnosti i fleksibilnosti koje nude, gde najjednostavniji u ponudi imaju virtuelne mašine i nekoliko pratećih usluga, dok npr. Amazonov *AWS* korisnicima omogućava korišćenje preko 200 servisa [1]. Jasno je da različite stvari važe za različite podskupove ovih platformi i ako se želimo detaljnije posvetiti izučavanju nekog dela, moramo se ograničiti na jedan deo ove ponude. Tip usluge na koji ćemo se mi ovde fokusirati se naziva računarstvo u oblaku.

U ostatku ove glave pokušaćemo da ukratko opišemo šta je to računarstvo u oblaku i šta možemo očekivati od istog. Sa ovim znanjem, u poslednjem odeljku ove glave ćemo detaljnije obrazložiti postupak i krajnji cilj ovog rada.

1.1 Osnovni koncepti okruženja u oblaku

Kako bi dalja priča o računarstvu u oblaku (eng. *cloud computing*) imala smisla, potrebno je uvesti neke definicije. Jedna od opšteprihvaćenih definicija ovog pojma dolazi od američkog Nacionalnog instituta za standarde i tehnologiju (*National Institute for Standards and Technology*, NIST) koji računarstvo u oblaku definiše kao model računarstva koji mora da ispuni narednih pet karakteristika [10]:

1. Samoposluživanje na zahtev: korisnik mora da ima mogućnost da samostalno obezbedi bilo koji računarski resurs po potrebi, bez ljudske intervencije na strani snabdevača (eng. *provider*, u ovom kontekstu to bi bila kompanija od koje iznajmljujemo resurse). Na primer, korisnik može klikom na dugme da dobije virtuelnu mašinu na korišćenje, bez ikakvog čekanja i posebnih procedura.

2. Širok mrežni pristup: sve mogućnosti su dostupne preko mreže i dostupne su različitim, heterogenim, klijentima. Ovo danas najčešće znači povezanost na internet i podrška za standardne prokole koji se koriste na internetu.

3. Udruživanje i deljenje resursa: snabdevač svoje resurse udružuje na nekoliko lokacija, a zatim ih deli među korisnicima. Korisnici nemaju mogućnost da vide resurse drugih korisnika, niti da znaju tačno gde se njihov kod izvršava ili njihovi podaci skladište, međutim mogu da odrede okvirnu lokaciju (npr. državu ili region).

4. Brza elastičnost: sve mogućnosti se mogu brzo skalirati u skladu sa potrebama, često uz mogućnost da se ovo odradi automatski. Na primer, u periodima kada ima mnogo saobraćaja, računarski resursi se mogu (automatski) povećati, a kada tražnja opadne, resursi se mogu smanjiti. Iz perspektive korisnika, mogućnosti deluju neograničeno i mogu se povećati u bilo koje doba (ovo naravno nije tačno iz ugla snabdevača: njihov posao je da obezbede dovoljno fizičkih resursa da održe ovaj privid).

5. Merenje potrošnje: potrošnja svakog resursa se može meriti u odgovarajućoj jedinici i na adekvatnom nivou apstrakcije, npr. koliko sati je virtuelna mašina uključena, koliko gigabajta diska je korisnik rezervisao, koliko gigabita saobraćaja je korisnik iskoristio i sl. Ova potrošnja je vidljiva korisniku i on može postavljati ograničenja.

NIST dalje definiše i tri modaliteta pružanja usluga: korišćenje snabdevačovih aplikacija na njihovoj infrastrukturi, gde korisnik samo podešava željeno ponašanje (softver kao servis — SaaS); korišćenje snabdevačove platforme za puštanje u rad korisnikovih aplikacija, gde korisnik nema kontrolu ni nad kakvom infrastrukturom, već samo predaje kod ili izgrađenu aplikaciju (platforma kao servis — PaaS); korišćenje i upravljanje snabdevačovom infrastrukturom, najčešće na nekom nivou apstrakcije (infrastruktura kao servis — IaaS). Poslednji modalitet zahteva najviše posla od strane korisnika, ali je i najfleksibilniji i najpopularniji za iole veće sisteme. Svaki veći snabdevač infrastrukture u oblaku nudi kombinaciju PaaS i IaaS servisa.

Naposletku, uvedimo još jednu podelu u odnosu na to ko je korisnik hardvera. Velike organizacije mogu zakupiti ili čak otkupiti cele data centre i na taj način obezbediti da se samo njihov sistem izvršava na konkretnim mašinama: ovo je recimo korisno u kontekstu veoma osetljivih podataka i pitanja npr. državne bezbednosti i ovakva okruženja se nazivaju „privatni oblak". Međutim, daleko je dominantniji „javni oblak", gde aplikacije i podaci različitih korisnika nisu striktno fizički (ali jesu logički!) razdvojeni. Postoje i razne varijante između ove dve krajnosti, tzv. „hibridni oblak", gde je jedan deo infrastrukture privatan, a drugi javni. Ova podela nije u toj meri suštinska kao prethodne dve i najčešće nema velikog uticaja na programerski deo posla, ali mi ćemo kada pričamo o „oblaku" do kraja rada uvek misliti na javni.

1.2 Snabdevači i servisi u javnom oblaku

Data definicija okruženja u oblaku je i dalje dovoljno široka da obuhvati veliki broj modernih snabdevača i iznajmljivača logičke infrastrukture, gde su najveće razlike u tome koliko su servisi u praksi elastični i koliko je ta elastičnost automatska, s nešto manjim naglaskom na merenje i ograničavanje potrošnje svakog resursa ponaosob. Međutim, bilo bi pogrešno posmatrati ih kao u potpunosti jednake. Ono što izdvaja bolje snabdevače okruženja u oblaku su količina, fleksibilnost, integracija i cena servisa koje nude.

Najpopularniji je *Amazon Web Services* (AWS), koji je i pionir u ovoj oblasti, zatim Majkrosoftov *Azure*, pa nešto manje korišćen *Google Cloud Platform* (GCP). Među manjim igračima se nalaze i Alibaba, Oracle i IBM [15]. Sva navedena okruženja imaju sličan skup servisa (najčešće pod različitim imenima) i data centre širom sveta koji mogu opsluživati korisnike. Svaki snabdevač naravno ima i svoje specifičnosti, što na nivou cele platforme, to i na nivou svakog pojedinačnog servisa. U

svakodnevnom govoru, kada pričamo o „oblaku" (ili „klaudu"), praktično uvek mislimo na jednog od ovih „mejnstrim" snabdevača i mi ćemo se ovde fokusirati na njih.

Sem osobina servisa, nešto manji diferencijator je rasprostranjenost i izolacija data centara koju snabdevač ima na raspolaganju. Svaki veći snabdevač upravlja hardverom u desetinama regiona širom sveta. Svaki region je podeljen na zone, međutim razlikuje se kako svaki snabdevač definiše šta je zona: AWS garantuje da su različite zone fizički udaljene jedne od drugih, dok recimo GCP i Azure ne pružaju te garancije, tako da katastrofičan događaj (npr. poplava, požar, zemljotres, ili samo dovoljno dugačak nestanak struje) na jednoj lokaciji može da obori ceo region [12].

Jedan način da obezbedimo da naš sistem uvek ima dostupan hardver na kom može da se izvršava je upotreba više snabdevača istovremeno (tzv. *multicloud* postavka), međutim ovo uvodi nov nivo kompleksnosti, pre svega što se tiče integracije između različitih snabdevača, i dodatne troškove i na strani korisnika i snabdevača (koje na kraju, naravno, plaća korisnik). Ovaj pristup ima smisla koristiti isključivo u slučajevima kada smo sigurni da jedan snabdevač ne može ispuniti naše zahteve na dovoljno dobar način, što je danas rezervisano samo za velike organizacije sa izuzetno specifičnim potrebama.

Osnovni servisi javnog oblaka

Spomenuli smo već da svi bolji snabdevači imaju sličan skup servisa koje nude. Glavna podela ovih servisa je na one koji nude računarske resurse ili neke hardverske primitive (IaaS servisi) i na one koji nude aplikativna rešenja kojima snabdevač upravlja za klijenta (PaaS servisi). Prvi se ugrubo mogu podeliti na servise koji nude virtuelne mašine, one koji pružaju resurse za skladištenje i mrežne servise. Od aplikativnih servisa, praktično uvek se podrazumeva korisnički sistem i servisi za upravljanje kontrolom pristupa, kao i (makar neke) baze podataka. Takođe, svaki snabdevač će nuditi i skup dodatnih aplikativnih servisa koji se najčešće neće u potpunosti preklapati ni sa jednim od konkurentskih platformi, npr. servise za prosleđivanje događaja i redovi (*SQS* na AWS-u, ili *CloudTasks* i *PubSub* na GCP-u), servisi za skladištenje i analizu velikih podataka (*Redshift* na AWS-u, *BigQuery* na GCP-u), brojni servisi za treniranje i upotrebu modela mašinskog učenja (npr. *Vertex AI*, *Natural Language AI* i *Vision AI* na GCP-u, *Q*, *Bedrock* i *Rekognition* na AWS-u, *Azure AI* i *OpenAI* servisi na Azure-u), itd.

Okosnica svake platforme su računarski servisi i ono što praktično svaki (IaaS) snabdevač nudi je mogućnost korišćenja virtuelnih mašina. Korisnik ima mogućnost da zada računarske mogućnosti koje će virtuelna mašina imati na raspolaganju, najčešće odabirom jedne od predefinisanih kombinacija procesorskih resursa i radne memorije koju će mašina imati na raspolaganju. Ova kombinacija se najčešće naziva „oblik" (eng. *shape*) i njene glavne osobine su broj CPU-ova i količina radne memorije ima na raspolaganju, npr. 0.5 CPU / 1GB RAM znači da će mašina deliti svoje procesorsko jezgro s nekim (tj. garantovano joj je svega pola sekunde procesorskog vremena po sekundi realnog) i imati 1GB radne memorije koje će moći da koristi. Konkretna apstrakcija koja se koristi za procesorske resurse zavisi od snabdevača, a nekad i od arhitekture procesora koja se koristi. Postoje različiti tipovi oblika koji su namenjeni poslovima koji intenzivnije koriste procesor ili memoriju i oni mogu biti fleksibilni, ali uvek postoji gornja granica na odnos CPU / GB RAM (uglavnom između 1/4 i 1/64). Sem kapaciteta hardvera, korisnik bira i operativni sistem koji želi da koristi, a koji će biti postavljen na mašini. Pristup mašini je najčešće moguć putem SSH protokola.

Virtuelne mašine nisu jedini široko rasprostranjeni računarski servis na platformama u oblaku. Praktično svi veći snabdevači nude i neki servis koji omogućava *serverless* računarstvo, tj. izvršavanje programskog koda bez upravljanja hardverom na kom se izvršava. U ovom modelu, korisnik piše funkcije ili manje programe koji se predaju platformi, a snabdevač obezbeđuje da se one imaju adekvatan hardver na kom se izvršavaju. Ovo značajno pojednostavljuje posao korisniku, ali mu takođe smanjuje fleksibilnost i povećava račun. Treća opcija je izvršavanje kontejnera, gde se snabdevaču predaju kontejneri, a on ih izvršava ili na hardveru kojim on upravlja ili koristeći resurse koje je korisnik prethodno zakupio. Jedna varijanta koja korisniku daje više kontrole je korišćenje Kubernetesa, o kom će biti više reči u sledećem odeljku.

Što se skladištenja podataka tiče, osnovna podela je na blokovska i objektna skladišta. Blokovska skladišta su ona koja emuliraju ponašanje blokovskih uređaja i najčešće se podrazumeva da imaju datotečni sistem na njima, tj. to su ona skladišta koja bismo nazivali „diskovima" u svakodnevnom govoru. Objektna skladišta su hijerarhijska skladišta koja smatraju datoteke građanima prvog reda, najčešće im dodeljuju jedinstveni identifikator, putanju, uglavnom podržavaju verziranje (tako da može postojati više verzija iste datoteke tj. objekta), mogućnost da im se pridruže metapodaci u obliku niza ključ-vrednost parova i dobru integraciju sa korisničkim

sistemom koji snabdevač pruža. Servisi koji pružaju objektno skladište uglavnom omogućavaju jednostavno objavljivanje ovih objekata na internet, tako da je jednim klikom moguće učiti deo hijerarhije javno dostupnom na nekoj veb adresi.

Kada nam je potrebno neko struktuiranije skladište, od svakog snabdevača infrastrukture u oblaku možemo dobiti neku SQL ili NoSQL bazu na korišćenje. Za razliku od prethodnih, ovaj skup servisa spada u PaaS modalitet, jer o samoj infrastrukturi brine snabdevač. Naravno, bazu možemo i sami postaviti na nekoj od virtuelnih mašina i nekom disku, ali prednosti korišćenja baza kojima upravlja snabdevač je to što ne moramo brinuti o dostupnosti i replikaciji podataka, kao ni ažuriranju same baze, što može biti veliki teret u većim sistemima (mane su već očekivana nešto manja fleksibilnost i veća cena). Osim klasičnih baza, najčešće možemo iznajmiti i neku bazu koja čuva podatke u memoriji koja bi služila kao sistem za keširanje, ali i neku bazu koja je namenjena za čuvanje, arhiviranje i analitiku ogromnih podataka. Takve analitičke baze su često specifične za konkretne snabdevače i nisu dostupne na drugi način (npr. Guglov *BigQuery* ili Amazonov *RDS*).

Mrežni servisi podrazumevaju sve od pravljenja virtuelnih mreža i podmreža (sabneta) za interno umrežavanje mašina, VPN i *peering* servisa, *firewall*-ova, kao i mreža za distribuciju sadržaja (eng. *Content Delivery Network* — CDN). Način na koji su ovi servisi implementirani zavisi od snabdevača.

Na kraju, nešto što se često podrazumeva i ne naglašava, svako okruženje u oblaku ima i relativno kompleksan korisnički sistem koji omogućava da se pristup svakoj pojedinačnoj akciji dozvoli ili zabrani konkretnim korisnicima ili grupama. Ovo važi kako za korisnike koji su ljudi, tako i za servisne naloge koje koriste aplikacije kako bi mogle da koriste neki od servisa u oblaku. Ovi sistemi su skoro univerzalno poznati pod akronimom IAM — *Identity and Access Management*. Sama implementacija i logika varira u zavisnosti od snabdevača[1].

Kubernetes

Vratimo se na trenutak na način puštanja aplikacija u rad. Tradicionalno, u kontekstu veb aplikacija, ovo znači pokretanje nekog procesa na (virtuelnoj ili fizičkoj) mašini koji će da sluša na nekom portu i servira sadržaj. Ovaj pristup ima nekoliko problema. Prvo, da bi se ovaj proces automatizovao, konvencionalan pristup je pod-

[1]Zanimljiv pregled kako ovo funkcioniše na AWS-u i GCP-u i jedno tumačenje kako je istorijski kontekst doprineo da Amazon razvije jedan od (nepotrebno) najkompleksnijih sistema ove prirode, je dostupan u ovoj objavi: `https://infosec.rodeo/posts/thoughts-on-aws-iam/`

razumevao skup skripti koje bi puštale aplikacije u rad, koje su teške za održavanje. Drugo, najčešće su ovi procesi i njihovi resursi delili mašine bez posebne izolacije, pa ako je neki proces maliciozan ili ranjiv, on može ugroziti sve ostalo što se dešava na toj mašini. Treće, uglavnom je čovek taj koji mora da vodi računa koliko instanci aplikacija postoji i gde su one puštene u rad (šta ako padne virtuelna mašina? Šta ako padne cela fizička mašina, ili cela zona ili region u oblaku?) Možda najvažnije, obezbeđivanje elastičnosti u praksi postaje jako teško: na koji način povećati kapacitet (pokrenuti dodatne instance) i isti smanjiti, ali tako da se one zapravo uključe u rad sistema i budu vidljive i dostupne svim ostalim delovima?

Neki od ovih problema se mogu rešiti upotrebom kontejnera: njihovo puštanje u rad je relativno standardizovano i oni podrazumevaju određeni nivo izolacije. Međutim, kako bi se obezbedilo nesmetano funkcionisanje više instanci na raznovrsnom nepouzdanom hardveru u svim okolnostima, potreban nam je sistem koji bi upravljao tim kontejnerima i svime što njima treba. Jedan takav sistem je Kubernetes. To je sistem koji je Gugl interno koristio za *orkestraciju* kontejnera (eng. *container orchestration*), tj. upravljanje njihovim životnim ciklusima, a koji su objavili kao softver otvorenog koda 2014. godine [9]. Danas svi veći snabdevači okruženja u oblaku nude upravljanje Kubernetes klasterom za korisnike kao servis, a neki imaju i besplatne opcije. Ne bi bilo u potpunosti adekvatno ovo nazivati „osnovnim" servisom, ali s obzirom na njegovu rasprostranjenost i činjenicu da ćemo ga koristiti u primerima u ostatku rada, posvetićemo mu narednih par strana. Ništa nas ne sprečava da koristimo Kubernetes van okruženja u oblaku, na bilo kojim mašinama, ali tu mogućnost nećemo razmatrati u ovom radu.

Osnovni cilj Kubernetesa je da apstrahuje hardver i dozvoli korisnicima da razmišljaju samo u kontekstu puštanja kontejnera i drugih resursa u rad, dok se on bavi raspoređivanjem posla. Sastoji se od dva dela: kontrolne ravni koja upravlja sistemom (ukoliko koristimo Kubernetes servise snabdevača u oblaku, ovime upravlja snabdevač) i mašina na kojima se izvršavaju kontejneri (eng. *worker nodes*, u daljem tekstu „čvorovi" klastera) što su u praksi virtuelne mašine u okruženju u oblaku. Kada kažemo da „Kubernetes nešto radi", uglavnom mislimo na logiku koja se izvršava u kontrolnoj ravni; čvorove možemo posmatrati kao računarske resurse za izvršavanje aplikacija[2] i po pravilu ne čuvaju nikakvo stanje. Ceo sistem se naziva i „Kubernetes klaster" s obzirom da je u pitanju klaster mašina kojima upravlja

[2]Ovo nije u potpunosti tačan model, jer postoje sistemske aplikacije koje se izvršavaju na čvorovima, ali je dovoljno blizu za naše potrebe ovde. Za detaljniji pregled, pogledati zvaničnu dokumentaciju: `https://kubernetes.io/docs/concepts/overview/components/`

Kubernetes.

Način na koji nešto puštamo u rad je pravljenje odgovarajućeg objekta u Kubernetes klasteru. Svaki objekat ima *specifikaciju* koju zadajemo i trenutni *status*. Jedan od osnovnih zadataka Kubernetesa je da obezbedi da status u svakom trenutku odgovara specifikaciji koju smo zadali. Sem toga, sastavni deo objekta su njegov tip (`apiVersion` i `kind`) i metapodaci koji moraju sadržati ime i imenski prostor u kom treba napraviti objekat, a opciono sadrže i nizove ključ-vrednost parova koji predstavljaju labele ili anotacije. Ovi objekti se po pravilu definišu u YAML formatu, tradicionalno kao datoteke koji se prosleđuju `kubectl` CLI alatu, ali se pomoću `kubectl`-a mogu i definisati i primeniti direktno bez upotrebe datoteka. Kubernetes nudi i API za sve operacije i to je interfejs koji ćemo implicitno koristiti u primerima u ovom radu, ali on nije ni približno zgodan za direktnu ljudsku upotrebu.

Konačno možemo odgovoriti na pitanje kako pokrenuti kontejner u Kubernetesu: potrebno je napraviti objekat tipa *Pod* koji će sadržati putanju do slike od koje želimo napraviti kontejner. Njegova (minimalna) definicija bi u YAML formatu izgledala ovako:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: default
spec:
  containers:
  - name: nginx-container
    image: nginx:stable-alpine3.17-slim
```

Pravljenjem ovog objekta u klasteru (npr. komandom `kubectl apply -f putanja-do-datoteke.yaml`) kažemo Kubernetesu da želimo da pokrene jednu instancu kontejnera baziranog na slici `nginx stable-alpine3.17-slim`. On zatim pronalazi čvor klastera koji ima slobodnih resursa i pokreće kontejner na njemu (generalno je dobra praksa definisati zahtevane resurse u specifikaciji, ovde ih izostavljamo zbog jednostavnosti). Na osnovu specifikacije možemo zaključiti da jedan pod može da sadrži više kontejnera, što se može koristiti za neku posebno komplikovanu logiku za inicijalizaciju, ili ako nam je potreban neki pomoćni proces koji je

usko vezan za drugi kontejner koji se izvršava[3].

Primetimo da i dalje nismo mnogo odmakli od pukog pokretanja kontejnera na nekoj mašini. Uistinu, „klot" podovi se vrlo retko koriste u praksi, sem u svrhe debagovanja. Mnogo korisniji i praktičniji objekti su *Deployment*-i i *StatefulSet*-ovi, koji nam omogućavaju da u specifikaciji zadamo broj replika podova koji želimo da pokrenemo i koji kontinuirano održavaju ovaj broj. U tim slučajevima takođe ima smisla definisati i *topology spread constraint*[4], kojim možemo definisati koji broj ili proporciju podova želimo rasporediti na čvor klastera koji ispunjava neke uslove (npr. raspoređivanje svih podova na isti čvor bi bilo katastrofalno u slučaju da on padne; isto tako, raspoređivanje svih podova na čvorove u istoj zoni bi bilo katastrofalno u slučaju da cela zona padne).

Drugi bitan koncept u Kubernetesu je umrežavanje. Kubernetes automatski dodeljuje internu IP adresu svakom podu, pomoću koje može da komunicira sa svim drugim podovima u klasteru. Međutim, ako pod iz bilo kog razloga umre, kada se ponovo napravi, on može dobiti novu IP adresu. Da bismo obezbedili neku stabilnu adresu, moramo napraviti objekat tipa *Service*. On predstavlja mrežnu primitivu u Kubernetes kontekstu i ponaša se kao raspoređivač opterećenja (eng. *load balancer*) nad podovima. Sem toga, on pravi i unos u Kubernetesovom internom DNS-u, tako da svaki pod razrešava `ime-servisa.imenski-prostor` na IP adresu servisa, preko koga se može doći do nekog od podova. Adresu servisa možemo zadati kao deo specifikacije, ako nam je bitno da ona bude ista čak i ako se servis ponovo napravi (za razliku od ponovnog pravljenja podova koje je normalno i očekivano, ovo za servise nije tipično, ali postoje situacije u kojima se dešava). Deo specifikacije servisa je *selektor* koji se koristi da definiše skup labela poda na koji se odnosi; svi podovi koji su označeni ovim labelama će dobijati saobraćaj ovog servisa.

Podrazumevani tip servisa je *ClusterIP*, što označava da on uzima samo internu IP adresu (ali iz zasebnog opsega, ne istog kao IP podova) i dostupan je isključivo unutar klastera. Međutim, servis se može definisati i sa tipom *LoadBalancer*, koji će od snabdevača zatražiti raspoređivač opterećenja i rutirati njegov saobraćaj na sebe. Ovo je najjednostavniji način za javno izlaganje nekog poda i omogućava našoj aplikaciji da prima saobraćaj s interneta. Pomoću anotacija na servisu (podsetimo se,

[3]Zvanična dokumentacija daje malo više detalja o mogućim scenarijima: `https://kubernetes.io/docs/concepts/workloads/pods/#how-pods-manage-multiple-containers`.

[4]`topologySpreadConstraints` je deo Pod API-ja, ali nema mnogo smisla koristiti ga samostalno. Videti `https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/`.

svaki objekat može sadržati skup labela i skup anotacija) možemo snabdevaču „preneti" kakav raspoređivač želimo, npr. možda želimo samo L4 (TCP / UDP) umesto L7 varijantu, ili želimo da se on nalazi u nekom konkretnoj podmreži (koja je npr. dostupna samo interno). Anotacije su generalno vezane za snabdevača i njihovi nazivi i skup mogućnosti veoma variraju. Tip servisa može biti i *NodePort* što otvara neki port na čvoru (generalno ne mnogo korisno i/ili vrlo loša praksa) ili *ExternalName* što saobraćaj mapira na DNS ime umesto na neki pod (nije kompatibilno sa HTTP protokolom), ali ova dva se u praksi vrlo malo koriste.

Pravljenje novog raspoređivača opterećenja na strani snabdevača za svaku aplikaciju nije mnogo logično ni ekonomično, ali i za to Kubernetes ima rešenje: *Ingress* objekat. U njegovoj specifikaciji možemo definisati konkretne *hostname*-ove (najčešće javne domene) i/ili putanje i pravila pod kojima se konkretan zahtev rutira na neki konkretan (uglavnom *ClusterIP*) servis. Da bi *Ingress* objekti bili korisni, potrebno je u klaster postaviti neki upravljač ulaznog saobraćaja (eng. *ingress controller*). Ovaj upravljač će služiti kao „glavni" raspoređivač opterećenja i napraviće jedan *LoadBalancer* servis koji će biti dostupan javno, a on će na osnovu svih *Ingress* objekata u klasteru odlučivati na koji servis će rutirati svaki konkretan zahtev. Postoje upravljači koji su bazirani na tradicionalnim obrnutim proksijima poput *nginx*-a ili *HAProxy*-ja, ali i aplikacija koje su pravljene konkretno za Kubernetes kao što su *Istio*, *Emissary* ili *Kong*.

Ovime smo pokrili tipove Kubernetes objekata koji se najčešće koriste koristite u jednostavnijim aplikacijama, ali smo ceo skup tek zagrebali. Postoje posebni objekti za čuvanje podešavanja u obliku ključ-vrednost parova (*ConfigMap* i *Secret*), za podešavanje i korišćenje trajnih diskova, tj. blokovskog skladišta okruženja u oblaku (*PersistentVolume* i *PersistentVolumeClaim*), servisnih korisnika i njihovih dozvola u klasteru (*ServiceAccount*, *Role* i *RoleBinding*) i raznih manje korišćenih načina za pokretanje procesa u klasteru (*Job*, *CronJob*, *ReplicaSet*, *DaemonSet*). Dodatno, korisnik i/ili aplikacije koje se koriste u klasteru imaju mogućnost definisanja dodatnih tipova objekata i dodeljivanja posebnog značenja njima, što čini ovaj skup praktično neograničenim. Neke od ovih ćemo videti do kraja rada.

Ono što Kubernetes projekat posebno izdvaja od drugih projekata otvorenog koda je konstantan razvoj i unapređivanje, ali i vrlo opsežna dokumentacija i podrška zajednice. Ne samo da su svi koncepti objašnjeni na vrlo pristupačan način, kako na visokom nivou tako i kroz konkretne primere, nego je uglavnom lako naći i dizajn dokumente i konkretne razloge zašto su određeni delovi sistema dizajnirani baš na

taj način. Zainteresovanog čitaoca ohrabrujemo da sve pojmove koji su nedovoljno detaljno objašnjeni u ovom odeljku pretraži na internetu, a ako želi da bolje razume bilo koji deo, upućujemo ga na zvaničnu dokumentaciju koja će uvek sadržati najrelevantnije i najsvežije informacije: `https://kubernetes.io/docs/home/`. U ostatku rada, a posebno u trećoj glavi, ćemo proći većinu ovih stvari kroz konkretne primere.

Izbor snabdevača za okruženje u oblaku

S obzirom da svi veći snabdevači imaju sličan osnovni skup servisa, izbor se svodi na specifične potrebe projekta: na primer, ako je od presudne važnosti koristiti Orakl bazu, njihovo okruženje u oblaku će imati prednost; ako je Guglov *BigQuery* najadekvatniji izbor za skladištenje velikih podataka, GCP je verovatno najbolji izbor; s druge strane, ako je bitno imati na raspolaganju najveći mogući broj data centara širom sveta koji su međusobno dobro izolovani, Azure i AWS će imati prednost. Naravno, svaki snabdevač će imati i svoj cenovnik i u zavisnosti od veličine firme i projekta će biti voljan da daje garancije i popuste u različitom opsegu i pod različitim uslovima.

Infrastrukture koje ćemu praviti u ovom radu neće zahtevati nikakve specijalne mogućnosti i neće biti naročito kompleksne u poređenju sa infrastrukturama koje koriste velike korporacije. S obzirom da su glavni koncepti dovoljno slični među snabdevačima, u ovom radu ćemo raditi na *Oracle Cloud Infrastructure* (OCI) platformi koja ima najvelikodušniju besplatnu ponudu. Jedan od ciljeva rada će biti da napravimo potpuno funkcionalnu infrastrukturu koja je pogodna za korišćenje od strane više različitih timova, koristeći samo besplatne resurse koje Orakl nudi.

Postupak i cilj rada

Infrastrukturu u oblaku je moguće postaviti i njome upravljati na više različitih načina. Prvi način na koji svaki korisnik naiđe i ujedno najprijemčiviji je korišćenjem grafičke konzole, tj. veb aplikacije koju svaki snabdevač nudi, gde možemo uneti tražene podatke u formu i klikom na dugme izvršiti neku akciju (npr. pravljenje virtuelne mašine koja će nam biti dodeljena). U ovom radu nećemo razmatrati ovaj pristup, već ćemo se fokusirati na opisivanje infrastrukture programskim kodom.

U narednoj glavi uvešćemo koncept infrastrukture kao koda (*Infrastructure as Code*, IaC), glavne razloge što je ovaj pristup poželjan i kroz jedan jednostavniji

primer predstaviti trenutno najpopularniji alat koji se koristi za ovu svrhu, Teraform. Videćemo da je ovaj alat praktično osnova za sve potencijalne „naslednike", tako da i dalje ima smisla posvetiti mu pažnju. U trećoj glavi ćemo prikazati noviji pristup, nešto bliži klasičnom programiranju koji koristi klasične programske jezike za opisivanje infrastrukture. Iako popularan, i dalje nije blizu dominaciji Teraforma, ali smatramo da itekako ima potencijala. Kao deo ove glave predstavićemo projekat koji se konceptualno nastavlja na primer iz druge glave i podržava funkcionalnosti koje nisu moguće upotrebom Teraforma. Cilj projekta je da predstavi na koji način se može povećati stepen automatizacije kada je u pitanju opisivanje i podešavanje infrastrukture, ali i svih pratećih servisa, upravo upotrebom alata za opisivanje infrastrukture programskim kodom.

Glava 2

Opisivanje infrastrukture kodom

Sve operacije na svim snabdevačima okruženja u oblaku se mogu izvršiti tako što unesemo odgovarajuće podatke i kliknemo na dugme u snabdevačovoj veb aplikaciji. Dodatno, veliki broj snabdevača ima i konzolnu aplikaciju kao alternatvni način upravljanja svim servisima. Čemu traženje još nekog dodatnog interfejsa za upravljanje infrastrukturom?

Ako radite za veliku korporaciju, verovatno postoji više timova čiji je posao vezan isključivo za upravljanje infrastrukturom. Kako bi ovi timovi mogli nesmetano da rade, pa čak i kako bi pojedinci u okviru istog tima ne bi „sudarali" na poslu, potrebne su procedure koje bi obezbedile da nema preklapanja, a koje usporavaju obavljanje posla. Ako uprkos njima do nekog preklapanja dođe, može da postane vrlo teško utvrditi ko je šta uradio i s kojim ciljem. Ovo potencijalno može imati katastrofalne ishode i upravljanje nekim velikim sistemom na efikasan i bezbedan način zahteva sistem za sebe koji nije lako učiniti kompatibilnim sa veb konzolama koje snabdevači nude.

Čak i ako zanemarimo sve probleme organizacione probleme, ni u slučaju da radimo sami obavljanje svih operacija kroz GUI ili CLI nije idealno. Ako 2019. godine rešimo neki problem nizom nekih akcija, a zatim 2024. prepoznamo da se nešto slično dešava, koja je verovatnoća da se setimo šta smo pre 5 godina uradili? Ako nam je to redovni posao, verovatno smo za pet godina rešili na stotine različitih problema i čak i da prepoznamo konkretan problem i imamo ideju o čemu se radi, vrlo teško ćemo se setiti svih koraka potrebnih za njihovo rešavanje.

Recimo da imamo savršenu kulturu dokumentovanja svega pa čak ni nalaženje rešenja za neke stare probleme nije veliki izazov (iako tako nešto zahteva dosta vremena na svakodnevnom poslu). Zamislimo da radimo na nekom problemu i is-

probavamo razna potencijalna rešenja na testnom okruženju. Nakon par dana rada i nekoliko desetina izvršenih operacija, konačno smo pronašli rešenje. Sada treba da to rešenje „prenesemo" na produkciono okruženje. Ali, nije dovoljno da je problem rešen i na produkciji, već treba i da održimo obećanje da su testno i produkciono okruženje praktično isti. Nakon desetina ovakvih „prenosa", gotovo sigurno će na testnom okruženju ostati neki neuspeli pokušaj rešavanja problema koji neće postojati na produkciji i samo je pitanje vremena kada će se neka aplikacija osloniti na tu razliku u podešavanjima i ili neće raditi (iako bi trebalo) ili, gore, radiće na testnom okruženju a neće na produkcionom. Doveli smo naše programere u zabludu i potencijalno ugrozili stabilnost celog sistema.

Primetimo da su analogni problemi prisutni i u programiranju, ali činjenica da radimo sa kodom i koristimo alate koji olakšavaju praćenje izmena i rad u timu, poput sistema za kontrolu verzija, značajno olakšava njihovo rešavanje. Kod jednoznačno opisuje ono što naša aplikacija radi i uvek možemo pregledati verzije koda u svakom trenutku u vremenu. Postavlja se pitanje što ne bismo koristili kod i da opišemo infrastrukturu?

Ako koristimo alate koji infrastrukturu opisuju kodom, trebalo bi da korišćenje grafičke konzole svedemo na minimum, kako bismo smanjili mogućnost za konflikte. Ne postoji funkcionalnost koja će zabraniti upotrebu nekog konkretnog interfejsa za pravljenje izmena u okruženjima u oblaku, ali očigledno je da opisivanje infrastrukture kodom, a zatim njeno menjanje na drugi način, ne vodi dobrom ishodu. S druge strane, verovatno ćemo s vremena na vreme koristiti veb interfejs da pregledamo trenutno stanje, ili da se informišemo o novim funkcionalnostima. Takođe, u izuzetnim slučajevima, npr. ako dođe do ozbiljnog incidenta gde je vreme rešavanja od izuzetnog značaja, možemo proceniti da je upotreba grafičkog alata efikasniji način rešavanja problema (ovo će zavisiti od konkretnog slučaja, ali i od našeg poznavanja i iskustva sa alatima za opisivanje infrastrukture kodom). U takvim situacijama upotreba alternativnog interfejsa može biti sasvim prihvatljiv izbor, dok god se na kraju pobrinemo da kod koji opisuje infrastrukturu odgovara realnom novonastalom stanju.

2.1 Kratak istorijat

Opisivanje infrastrukture kodom zapravo nije nova ideja: starija je od velikog broja okruženja u oblaku koje danas koristimo. Prva verzija alata *Puppet* je izašla

2005. godine i njen cilj je bio da deklarativno opiše podešavanja nekog sistema [16]. Pojavom konkurenata poput *Chef*-a i *Ansible*-a, ovo je prošireno i na upravljanje resursima drugih snabdevača, slično onome što danas podrazumevamo pod pojmom „infrastruktura kao kod".

Moderna primena ovog koncepta na snabdevače okruženja u oblaku počinje 2011. godine kada AWS uvodi *CloudFormation* servis koji omogućava opisivanje AWS infrastrukture pomoću YAML datoteka. Upravo *CloudFormation* postaje inspiracija za *Terraform*, alat čija je prva verzija objavljena 2014. godine i koji je inicijalno podržavao samo dve platforme (AWS i *DigitalOcean*) [7], a koji je danas dominantan za opisivanje bilo koje infrastrukture u oblaku (a i šire) pomoću deklarativnog programskog jezika koji je stvoren isključivo za tu svrhu.

Danas postoje i mnogi drugi alati koji mogu da posluže za obavljanje sličnih zadataka. Neki, poput Amazonovog *CloudFormation*-a i Guglovog *Deployment Manager*-a koriste YAML za opis infrastrukture i specifični su za konkretnog snabdevača (AWS ili GCP), dok je Majkrosoft razvio domenski-specifičan jezik Bicep koji se koristi za opis infrastrukture na njihovoj platform pomoću *Azure Resource Manager*-a. Kada je reč o alatima koji podržavaju više platformi, sem Teraforma, u upotrebi su i OpenTofu, alat otvorenog koda koji je nastao kao klon Teraforma nakon što je ovaj promenio licencu u manje permisivnu varijantu, i Pulumi koji ima nešto programerskiji pristup i o kome će više reči biti kasnije u ovom radu. Takođe postoje i noviji, eksperimentalniji pristupi, poput *NET Aspire* alata koji je deo .NET razvojnog okvira i (trenutno) nema ambicije da podrži sve funkcionalnosti koje tipično podrazumevamo kada spominjemo IaC, ali je značajno lakši za upotrebu od strane programera i podržava najčešće slučajeve upotrebe [8].

2.2 Teraform

Osnovni koncepti

Cilj Teraforma je opisivanje infrastrukture u oblaku kroz kod. Za te potrebe se koristi domenski specifičan jezik *HashiCorp Configuration Language* (HCL), nazvan po kompaniji koja je napravila ovaj alat, *HashiCorp*. Skup HCL datoteka se zatim prosleđuje CLI alatu `terraform`, koji ih čita i pravi odgovarajuće izmene na okruženju u oblaku kako bi realna infrastruktura odgovarala opisu.

Teraform održava trenutno stanje i pri svakom čitanju pravi samo neophodne

izmene kako bi doveo to stanje do željenog. Podrazumevano se to stanje nalazi u datoteci na korisnikovoj mašini u posebnom direktorijumu. Ovo nije preterano praktično ako radimo s drugim ljudima, već se u praksi koriste posebni *storage backend*-i koji određuju gde ćemo čuvati datoteke sa stanjem. Najčešći izbor za ovaj *backend* je objektno skladište odgovarajućeg okruženja u oblaku.

HCL jezik je agnostičan u odnosu na snabdevača. Međutim, jasno je da se iste operacije različito obavljaju na različitim okruženjima. Kako bi prevazišao ovaj problem, Teraform uvodi koncept *snabdevača* (eng. *provider*): njihov posao je da „prevode" ono što je izraženo HCL datotekama u API pozive ka odgovarajućoj platformi, tj. oni su logička apstrakcija API-ja konkretnih platformi. Trenutno postoji na hiljade različitih snabdevača za raznorodne platforme i nisu ograničeni samo na okruženja u oblaku [18].

HCL jezik

Sintaksa HCL-a je dosta jednostavna i može se predstaviti na sledeći način:

```
<TIP_BLOKA> "<OZNAKA_BLOKA>" "<OZNAKA_BLOKA>" {
  <IDENTIFIKATOR> = <IZRAZ>
  <IDENTIFIKATOR> = <IZRAZ>
  ...
}
```

HCL datoteke su nizovi ovakvih blokova. Osnovni tip bloka je *resource* koji definiše neki resurs infrastrukture. Prva oznaka označava tip resursa, dok druga označava ime pomoću kojeg se kasnije može referencirati. Svaki blok sadrži niz *atributa* u obliku niza ključ-vrednost parova; Teraform atribute naziva *argumentima*, a mi ćemo ova dva pojma smatrati sinonimima u ovom kontekstu. Na primer, na ovaj način se može opisati jedna instanca virtuelne mašine na Oraklovoj infrastrukturi u zoni *eu-frankfurt-3* (*availability domain* je Oraklov naziv za zonu):

```
resource "oci_core_instance" "vm_instanca" {
    availability_domain = "XiDj:EU-FRANKFURT-1-AD-3"
    compartment_id = "id_compartmenta"
    shape = "VM.Standard.E2.1.Micro"
}
```

Svi ostali tipovi blokova su „u službi" resursa, tj. imaju smisla samo ako ih neki resurs implicitno koristi ili referencira. Za razliku od resursa, neki od ovih tipova zahtevaju samo jednu ili nijednu oznaku. Navedimo ih i ukratko opišimo:

- *provider* blok definiše podešavanja snabdevača. Unutar specijalog *terraform* bloka možemo imati *required_providers* blok gde ćemo definisati samog snabdevača i njegovu verziju; ovo ćemo videti u narednom odeljku.
- *data* blok definiše neki izvor podataka iz kojeg možemo čitati. Na primer, možemo dobiti listu zona koji trenutni region podržava.
- *module* blok omogućava grupisanje više resursa i izvora podataka u jedan entitet.
- *variable* blok deklariše neku ulaznu promenljivu koja se može ili mora proslediti da bi naša podešavanja radila.
- *output* blok deklariše izlazne promenljive podešavanja: one se mogu koristiti da se nadovežu različita podešavanja ili da se na kraju izvršavanja ištampaju neki podaci na standardni izlaz (npr. IP adresa pomoću koje se može povezati na neku mašinu).
- *locals* blok definiše promenljive koje su lokalne za jednu izvornu datoteku i mogu se koristiti na više različitih mesta u njemu.

2.3 Opis jednostavne infrastrukture u Teraformu

Prođimo sad kroz opis jedne infrastrukture koja definiše osnovnu mrežnu postavku i podiže jedan Kubernetes klaster u Oraklovom okruženju u oblaku. Ceo kod je dostupan na GitHub repozitorijumu:
`https://github.com/luka-j/terraform-oci-k8s-simple`

Moduli

Fokusirajmo se prvo na `modules` direktorijum. Ovde se nalaze tri modula od kojih ćemo kasnije da izgradimo infrastrukturu.

Compartment

U OCI (*Oracle Cloud Infrastructure*) okruženju, svi servisi moraju pripadati nekom *compartment*-u iliti *odeljku*, tj. on predstavlja zajednički imenski prostor za ceo projekat. Stoga je logično da je prvi korak pravljenje ovog odeljka. Teraform izvorne datoteke koje ovo opisuju se nalaze u direktorijumu `compartment`.

Uobičajeno je razdvojiti deklaracije promenljivih od resursa, te je to i ovde urađeno. U datoteci `variables.tf` možemo naći ulazne promenljive, odnosno sve podatke koje ovaj modul zahteva. Deklaracija promenljive izgleda ovako:

```
variable "tenancy_id" {
  description = "OCID of the tenancy in which to create a compartment"
  type = string
}
```

Ova promenljiva definiše identifikator naloga unutar kojeg ćemo napraviti odeljak. U okviru *variable* bloka, *type* određuje ograničenje tipa i može biti ili jedan od *string*, *number*, *bool* ili kolekcija poput *list(<tip>)*, *set(<tip>)*, *map(<tip>)*, *object(<naziv_atributa> = <tip>, ...)* ili *tuple([<tip>, ...])*. Mi ćemo se zadržati na primitivnim tipovima. *description* je opcioni opis promenljive, ali je dobra praksa uvek ga definisati.

Na sličan način definišemo i promenljive koje određuju naziv i opis odeljka. Pošto želimo da promenljiva koja predstavlja opis bude opciona, u taj blok dodajemo i `default = ""` koji postavlja podrazumevanu vrednost ove promenljive na praznu nisku.

Pređimo sad na `main.tf` datoteku u kojoj su definisani resursi i izvori podataka. Na vrhu imamo blok koji definiše snabdevača:

```
terraform {
  required_providers {
    oci = {
      source = "oracle/oci"
      version = ">= 4.102.0"
    }
  }
}
```

Ovime određujemo da ova datoteka zahteva `oracle/oci` snabdevač koji će se podrazumevano tražiti na Teraformovom Registru, centralnom repozitorijumu za snabdevače i module (veb interfejs je dostupan na `https://registry.terraform.io`), najmanje verzije `4.102.0`. Konačno, definišimo resurs za odeljak, tipa`oci_identity_compartment` i nazovimo resurs (ne i odeljak!) `tf_compartment`:

```
resource "oci_identity_compartment" "tf_compartment" {
    compartment_id = "${var.tenancy_id}"
    description = "${var.description}"
    name = "${var.name}"
}
```

Ovde vidimo na koji način koristimo promenljive koje smo deklarisali u `variables.tf`: upotrebom sintakse `${}`, unutar `var` objekta možemo pristupiti vrednosti bilo koje promenljive. U ovom slučaju, ekvivalentno bi bilo i samo `var.tenancy_id` (bez upotrebe šablonske niske), razlika je isključivo u stilu. Ovde možemo istaći i jednu specifičnost OCI resursa, a to je da oni uvek imaju *compartment_id* polje, iako se za resurse koji ne pripadaju odeljku nego *tenancy*-ju očekuje da se tu prosledi identifikator *tenancy*-ja.

Definišimo i jedan izvor podataka koji će nam omogućiti da pročitamo zone (odnosno *availability domain*-e) za region u kom se nalazi nalog:

```
data "oci_identity_availability_domains" "ads" {
  compartment_id = "${var.tenancy_id}"
}
```

Za kraj, u `outputs.tf` datoteci definišimo izlazne promenljive koje će nam biti korisne u drugim modulima, konkretno identifikator odeljka i listu zona:

```
output "compartment_id" {
  value = oci_identity_compartment.tf_compartment.id
}

output "availability_domains" {
  value = data.oci_identity_availability_domains.ads.availability_domains
}
```

Na ovom primeru možemo videti i kako pristupamo podacima: ako ih definiše resurs, sintaksa je *tip_resursa.naziv_resursa.naziv_izlaza* (*id* je izlazna promenljiva

resursa koji definiše odeljak), a ako ih definiše izvor podataka potreban je prefiks *data.*.

VCN

Pređimo sad na pravljenje mreže, ili u OCI terminologiji *Virtual Cloud Network*-a (VCN). Prva stvar koja upada u oči u `modules/vcn` direktorijumu je veći broj datoteka. Deklaracije ulaznih i izlaznih promenljiva u `variables.tf` i `outputs.tf` nećemo posebno komentarisati, s obzirom da je logika potpuno analogna. U ovom delu očekujemo da na ulazu dobijemo identifikator odeljka (`compartment_id` promenljiva), naziv i labelu za DNS.

Krenimo od `main.tf` datoteke. Ispod već viđenog bloka koji definiše potrebne snabdevače, imamo novi blok `module "vcn" {}`. Na ovaj način kažemo Teraformu da hoćemo da napravimo modul koristeći izvorne datoteke sa određene lokacije i prosleđujemo mu neke promenljive. Ove promenljive modul vidi kao svoj ulaz. Ceo blok izgleda ovako:

```
module "vcn" {
  source  = "oracle-terraform-modules/vcn/oci"
  version = "3.5.3"
  compartment_id = "${var.compartment_id}"

  vcn_name = "${var.name}"
  vcn_dns_label = "${var.dns_label}"
  vcn_cidrs = ["10.0.0.0/16"]

  create_internet_gateway = true
  create_nat_gateway = true
  create_service_gateway = true
}
```

Argument *source* određuje lokaciju izvornih datoteka, podrazumevano se odnosi na identifikator u Teraformovom Registru. Stranicu ovog modula možemo videti na adresi `https://registry.terraform.io/modules/oracle-terraform-modules/vcn/oci/latest`. Kao ulazne promenljive mu prosleđujemo naziv, DNS labelu (sa našeg ulaza), listu opsega IP adresa koje pripadaju mreži i logičke promenljive koje

označavaju da želimo da napravimo *gateway*-e za pristup internetu i lokalno umrežavanje. To je to — napravili smo (najjednostavniju) virtuelnu mrežu na Oraklovoj infrastrukturi u oblaku.

Međutim, ono što zapravo želimo od mreže, je privatni i javni prostor. Sve[1] što se nalazi u privatnom prostoru je dostupno samo interno, drugim delovima te mreže, dok je sve u javnom dostupno svima putem interneta. Tipičan način da ovo ostvarimo je pravljenje podmreža (eng. *subnet*), jedne privatne i jedne javne. Ovo je definisano u datotekama `private-subnet.tf` i `public-subnet.tf`.

Obe datoteke definišu po jedan resurs tipa `oci_core_subnet`, a ovde ćemo prikazati samo kako izgleda definicija javne podmreže:

```
resource "oci_core_subnet" "vcn_public_subnet" {
  # Required
  compartment_id = "${var.compartment_id}"
  vcn_id = "${module.vcn.vcn_id}"
  cidr_block = "10.0.0.0/24"

  # Optional
  display_name = "${var.name}-public-subnet"
  route_table_id = "${module.vcn.ig_route_id}"
  security_list_ids = [oci_core_security_list.public_security_list.id]
}
```

U oba slučaja prosleđuju id odeljka, id mreže (iz izlazne promenljive `module.vcn.vcn_id`, sintaksa analogna onoj za izlaze izvora podataka), disjunktni blok IP adresa koji se dodeljuju podmreži i naziv koji se prikazuje u konzoli. Ono gde se ove podmreže značajnije razlikuju su preostali argumenti. Za tabelu rutiranja, javna podmreža koristi tabelu *internet gateway*-a koji je napravio VCN modul (`ig_route_id`), dok privatna koristi tabelu *NAT gateway*-a (`nat_route_id`). U narednoj glavi ćemo detaljnije razmatrati postavku mreže, ali ukratko, mašine s javnom IP adresom treba da koriste *internet gateway*, dok one bez javne adrese treba da budu iza nekog NAT sistema poput istoimenog *gateway*-a. Druga bitna razlika je u listama koje određuju koji saobraćaj se prima, a koji ne. U ovom delu referenciramo objekte koje još nismo spomenuli, `public_security_list` i `private_security_list`.

[1] mašine i raspoređivači opterećenja, uglavnom

Ovi resursi su definisani u (skoro) istoimenim datotekama, `public-security-list.tf` i `private-security-list.tf`. S obzirom da su nešto duži, ovde ih nećemo prikazivati u celosti. Kao i podmrežama, i njima se prosleđuju `compartment_id`, `vcn_id` i `display_name` sa istovetnim značenjem. Međutim, oni sadrže i niz `egress_security_rules` i `ingress_security_rules` blokova. Prvi određuju kakav saobraćaj sme da napusti mrežu, dok drugi opisuje saobraćaj koji sme da dođe do resursa koji pripadaju mreži. Ovo se opisuje opsegom IP adresa, protokolom i opciono nekim atributom protokola (npr. broj ili opseg portova za TCP). Paketi koji se ne uklapaju u ovaj opis se ignorišu (*drop*-uju) i pošiljalac nikad ne dobija odgovor na njih. U ovom primeru nećemo posebno ograničavati izlazni saobraćaj, pa će resursi u oba slučaja izgledati ovako:

```
resource "oci_core_security_list" "public_security_list" {
  compartment_id = "${var.compartment_id}"
  vcn_id = "${module.vcn.vcn_id}"
  display_name = "${var.name}-security-list-for-public-subnet"

  egress_security_rules {
      stateless = false
      destination = "0.0.0.0/0"
      destination_type = "CIDR_BLOCK"
      protocol = "all"
  }

  ingress_security_rules {
    ...
  }

  ingress_security_rules {
    ...
  }
  ...
}
```

U oba slučaja ćemo propustiti i deo ICMP saobraćaja jer ga OCI interno koristi. Suštinska razlika je u TCP saobraćaju: na javnoj podmreži propuštamo saobraćaj sa

svih adresa, dok na privatnoj samo onaj iz opsega `10.0.0.0/16`, tj. iz same mreže. Ovde možemo biti i nešto restriktivniji, pa se npr. ograničiti samo na portove 22 (ssh), 80 (http) i 443 (https), ali zbog jednostavnosti to izostavljamo. Takođe, ako znamo da se nalazimo iza nekog obrnutog proksija (npr. mreže za dostavljanje sadržaja), umesto svih adresa, na javnu podmrežu možemo propustiti samo one opsege koji pripadaju proksiju. Na privatnoj mreži ne bi valjalo ograničavati portove jer će se u njoj nalaziti Kubernetes čvorovi, a nismo sigurni šta će oni sad ili u budućnosti koristiti. Dodatno, u privatnoj mreži propuštamo i UDP saobraćaj koji potiče sa iste, pošto delovi Kubernetesa ili same aplikacije mogu komunicirati međusobno koristeći neki protokol baziran na UDP-u.

Ovime smo završili postavku mreže. S obzirom da se ovde susrećemo sa kodom koji je podeljen u nekoliko datoteka, pažljiv čitalac se može zapitati kako Teraform zna kojim redom da ih „izvršava"? Manje zadovoljavajuć odgovor bi bio: „S obzirom da je sve deklarativno, koncept izvršavanja zapravo ne postoji, Teraform ima svoje načine da uskladi traženo i realno stanje". Nešto detaljniji odgovor leži u načinu na koji se ovo usklađivanje stanja radi. Nakon što spoji sve `.tf` datoteke, Teraform interno pravi graf zavisnosti na osnovu ulaza i izlaza svakog resursa i modula i kreće od onih resursa koji nemaju nikakve zavisnosti. Ovo znači da Teraform može raditi i na više resursa istovremeno, što će se koristiti kao tehnika optimizacije. Ako zavisnost postoji ali nije izražena kroz upotrebu promenljivih, u svaki resurs ili modul se može dodati meta-argument `depends_on` koji zavisnost eksplicitno definiše (meta-argumenti su argumenti koji se mogu dodati u bilo koji resurs i kontrolišu njihovo ponašanje na neki način).

Kubernetes

Sada imamo sve preduslove da napravimo jedan Kubernetes klaster, čija podešavanja su definisana u direktorijumu `modules/kubernetes`. Ovaj modul zahteva nešto više promenljivih, uglavnom vezanih za mrežu i kapacitete klastera, koje ćemo detaljnije opisati po potrebi.

U datoteci `main.tf` imamo definicije tri resursa: `oci_containerengine_cluster`, `oci_containerengine_node_pool` i `oci_core_instance`. Prvi se odnosi na sam Kubernetes, tj. na njegovu kontrolnu ravan kojom će upravljati Orakl. Ovom resursu moramo proslediti ime, verziju i identifikatore odeljka i mreže (*VCN*-a). Dalje, podešavamo *Kubernetes endpoint*, odnosno ulaznu tačku za Kubernetesov HTTP API preko kojeg se može upravljati sistemom i koju drugi alati interno koriste,

narednim blokom:

```
endpoint_config {
  is_public_ip_enabled = false
  subnet_id = "${var.endpoint_subnet_id}"
}
```

U odnosu na to da li je ova ulazna tačka izložena javno (dostupna preko interneta) ili ne (dostupna samo na internoj mreži), Kubernetes klastere nazivamo *javnim* ili *privatnim*. Naravno, da bi se bilo koja akcija izvršila, neophodno je autentifikovati se, ali se javni klasteri svakako smatraju lošom praksom zbog nepotrebnog sigurnosnog rizika. Mi ćemo napraviti privatni klaster i smestiti njegovu pristupnu tačku u privatnu podmrežu koju smo napravili u prethodnom koraku u skladu sa najboljim praksama. Ovo znači da kada budemo želeli da pristupimo klasteru, moramo to učiniti sa interne mreže; više reči o tome će biti kroz par pasusa.

Dodatno, resursu koji opisuje klaster prosleđujemo i blok `options`, u kojem onemogućavamo neke dodatke koje nećemo koristiti (*Kubernetes Dashboard* i *Tiller*, deo *Helm v2* alata). Drugi deo `options` bloka je značajno važniji, jer u njemu dodeljujemo prostor IP adresa i podmrežu za servise:

```
options {
    ...
    kubernetes_network_config {
        pods_cidr = "10.244.0.0/16"
        services_cidr = "10.96.0.0/16"
    }
    service_lb_subnet_ids = ["${var.service_lb_subnet_id}"]
}
```

Ove opsege IP adresa će koristiti interna mreža samog Kubernetesa (ne VCN!) i servisi koji njima odgovaraju neće biti vidljivi van Kubernetes klastera. Ovo praktično znači da će do njih moći da dođu samo podovi, dok za sve mašine (čak i ako one fizički hostuju podove) oni neće biti dostupni. Ovi opsezi ne bi trebalo da se preklapaju ni sa jednim opsegom neke druge mreže i trebalo bi da sadrže dovoljno adresa za sve podove i servise koji će se pokretati u klasteru. Argumentom `service_lb_subnet_ids` određujemo gde će se praviti raspoređivači opterećenja, koji su jedan od OCI servisa, kada se u klasteru napravi *Service* tipa *LoadBalan-*

cer: ako želimo da ovi *Service*-i budu izloženi javno, ovde ćemo proslediti javnu podmrežu.

Kada se `oci_containerengine_cluster` podigne, možemo reći da imamo Kubernetes, pa čak i koristiti CLI alate poput `kubectl` da ispitamo njegovo stanje, međutim ne možemo ništa korisno uraditi s njim. Ako bismo pokušali da pokrenemo neki pod, on ne bi imao gde da se izvršava, pošto nema dostupnih čvorova. Ovaj problem rešavamo tako što ćemo napraviti skup virtuelnih mašina i povezati ih sa napravljenim klasterom, što se u Teraformu opisuje resursom `oci_containerengine_node_pool`. Sem standardnih argumenata koje smo videli u prethodnom delu i identifikatora klastera, imamo još nekoliko blokova koje ćemo ukratko opisati:

- `node_config_details` određuje veličinu klastera (`size`) i u kojim zonama će se nalaziti (`placement_configs`). Uvek bi trebalo imati mašine u više zona, za slučaj da neka od njih padne.

- `node_shape` i `node_shape_config` određuju oblik mašine: mi ćemo koristiti ARM procesore, stoga oblik `VM.Standard.A1.Flex` a broj procesora i veličinu memorije uzimamo iz argumenata. Iako ARM nije tipičan izbor, danas ga dosta (modernog) softvera podržava, a za ovaj tip Orakl ne naplaćuje prva 4 CPU-a i 24GB RAMa.

- `node_source_details` opisuje sliku (operativnog sistema) koju želimo na mašinama, čiji identifikator smo prethodno našli na zvaničnoj stranici `https://docs.oracle.com/en-us/iaas/images/`, kao i veličinu diska gde upisujemo minimalnu vrednost 50 (bilo koja manja vrednost bi dovela do greške pri izvršavanju).

- `initial_node_labels` su opcione oznake čvorova, koje kasnije mogu da se referenciraju za potrebe aplikacija koje se izvršavaju u Kubernetesu i tu definišemo samo ime.

Onog trenutka kada `oci_containerengine_node_pool` postane dostupan, konačno imamo funkcionalan klaster. Ostaje još jedan problem koji smo prethodno spomenuli: trenutno nemamo nijedan način da mu pristupimo. Za ove potrebe, napravićemo virtuelnu mašinu u *javnoj* podmreži koja će biti izložena preko interneta, kojoj ćemo moći da pristupimo preko SSH protokola. Podsetimo se da resursima u privatnoj podmreži mogu da pristupe svi koji su na istoj *mreži* (uključujući i javnu

podmrežu), a javnoj podmreži ceo internet. Ova mašina se često naziva *bastion host* ili samo *bastion* i predstavlja jedinstvenu tačku pristupa za sve resurse na internoj mreži.

Virtuelnu mašinu u Teraformu definišemo pomoću `oci_core_instance` resursa, koji je sličan prethodnom *node_pool*-u. Ovde kao oblik biramo najmanju moguću instancu sa procesorom AMD64 arhitekture, `VM.Standard.E2.1.Micro`, čije su prve dve instance besplatne. Ovo nas ograničava na *availability domain* `XiDj:EU-FRANKFURT-1-AD-3` s obzirom da ove instance nisu dostupne u drugim zonama. Preostaje nam da objasnimo još dva dela koja ranije nismo videli:

```
create_vnic_details {
    subnet_id = "${var.service_lb_subnet_id}"
}
metadata = {
    ssh_authorized_keys = "${var.ssh_public_key}"
}
```

Prvim blokom određujemo u koju podmrežu želimo da stavimo ovu virtuelnu mašinu, što će biti ista kao i podmreža u kojoj će biti izloženi Kubernetes *LoadBalancer* servisi, tj. u ovom konkretnom primeru javna. U drugom delu određujemo metapodatke za virtuelnu mašinu i jedini koji nam je potreban je `ssh_authorized_keys` koji određuje sadržaj istoimene datoteke gde će se nalaziti javni ključevi i njega postavljamo na naš javni ključ. Pomoću odgovarajućeg privatnog ključa ćemo moći da se ulogujemo na ovu mašinu preko SSH protokola.

Ako čitalac malo detaljnije pogleda prethodni blok koda, može se zapitati zašto prvi blok ne sadrži znak jednakosti, dok kod drugog on postoji? Odgovor leži u tome da su u pitanju različiti tipovi. `create_vnic_details` je *blok*, isto kao što su `resource` i `data` blokovi, koji je deo definicije ovog resursa i ne definiše nikakve labele (iako blokovi uvek mogu da definišu proizvoljan broj labela, kao što `resource` traži dva ili `module` jedan, za ugnježdene blokove je ovo retka pojava). S druge strane, `metadata` je običan argument tipa *object*, kojem dodeljujemo objektni literal definisan vitičastim zagradama i sadržajem unutar njih. Blokovi, poput klasa u objektno orijentisanim programskim jezicima, definišu svoj skup atributa (u ovom slučaju, između ostalih, `subnet_id`), dok su objekti nalik *map* ili *dictionary* strukturama u konvencionalnijim jezicima i mogu imati proizvoljne vrednosti ključeva.

Zaista, šta god prosledimo kao deo `metadata` objekta ćemo moći i kasnije da vidimo, ali `ssh_authorized_keys` atribut ima i specijalno značenje.

Pokretanjem bastion mašine smo završili postavljanje Kubernetes klastera. Ostaje još da spojimo delove opisane u prethodnih nekoliko odeljaka.

Spajanje modula i podešavanje snabdevača

Sada imamo sve neophodne delove da spojimo celu infrastrukturu, preostaje još da spojimo delove slagalice. Vratimo se par nivoa iznad, u `1-cluster/` direktorijum, tzv. *koreni* modul, pošto ćemo iz njega uključivati ostale module, a ništa neće uključivati njega. Kada pogledamo sadržaj `main.tf` datoteke, trebalo bi da prepoznamo neke nazive (konkretno, promenljivih) iz prethodnih odeljaka.

U *main* datoteci su definisana tri modula koja smo nazvali *compartment*, *vcn* i *kubernetes* (nazivi mogu biti proizvoljni). Svaki od njih ima argument *source* i putanju do direktorijuma sa datotekama gde su definisani njegovi resursi (npr. `../modules/vcn`) i niz argumenata koji će biti prosleđeni kao promenljive (koje smo, po konvenciji, deklarisali u odgovarajućim `variables.tf` datotekama). Putanja do datoteka sa resursima mora počinjati sa `./` ili `../`, jer se u suprotnom podrazumeva da želimo modul sa Teraformovog Registra. Izlaze modula prosleđujemo kao argumente drugim koristeći već poznatu sintaksu, pa tako npr. identifikatoru OCI odeljka pristupiti pomoću `module.compartment.compartment_id`.

Da bi sve ovo zapravo radilo, Teraform mora da zna kako da pristupi interfejsu OCI platforme. Ovo je definisano `provider` blokom koji smo smestili u `provider.tf` datoteci. Njemu kao argumente prosleđujemo identifikator *tenancy*-ja, našeg korisničkog naloga, region, kao i otisak i putanju do privatnog ključa koji koristimo za autentifikaciju. Konkretan postupak podešavanja snabdevača se razlikuje od snabdevača, a korisni resursi se uvek mogu naći na stranici odgovarajućeg snabdevača na Teraform Registru i/ili na stranici snabdevača usluga. U slučaju Orakla, uputstvo je dostupno na adresi: `https://docs.oracle.com/en-us/iaas/developer-tutorials/tutorials/tf-provider/01-summary.htm`.

Ostaje još da objasnimo kako se postavljaju promenljive u korenom modulu. Jedna opcija je korišćenjem `-var` opcije CLI alata, međutim ovo nije mnogo praktično. Umesto toga, pravimo datoteku `terraform.tfvars` koji će se sastojati isključivo od definicija promenljivih i koristi HCL sintaksu, npr. za običnu nisku `tenancy_ocid="ocid.abc"`. Upotrebom uglastih ili vitičastih zagrada se mogu definisati i liste i objekti, mada nam to u ovom primeru nije potrebno. Teraformov CLI će

automatski učitati ovu datoteku i postaviti promenljive na odgovarajuće vrednosti; moguće je koristiti i argument komandne linije `-var-file="putanja-do-datoteke"` u slučaju da se datoteka drugačije zove. S obzirom da ova datoteka sadrži osetljive podatke, nju ne dodajemo na Git.

Da bi smo primenili sva podešavanja o kojima smo diskutovali u ovom poglavlju, neophodno je da prvo inicijalizujemo Teraform u `1-cluster` direktorijumu komandom `terraform init`. Nakon što se preuzme i inicijalizuje Teraform snabdevač, trebalo bi da dobijemo poruku o uspehu. Komandom `terraform plan` instruišemo Teraform da pročita sva podešavanja i ispiše nam šta će da uradi. Ako smo sve dobro postavili, trebalo bi da dobijemo nešto duži izlaz, koji će da sadrži listu svih resursa (ne modula!) i onih argumenata koji su poznati, dok će ostali (većina njih) umesto vrednosti imati tekst `(known after apply)`. S obzirom da krećemo od prazne infrastrukture i samo pravimo resurse, poslednja linija plana treba da glasi:

```
Plan: 15 to add, 0 to change, 0 to destroy.
```

Imamo i opciju da sačuvamo ovaj plan kako bismo ga kasnije primenili. Nešto jednostavnije varijanta je da ga primenimo odmah, komandom `terraform apply`. Ova komanda će opet ispisati plan i tražiti od nas da potvrdimo da je to zaista ono što želimo da uradimo unosom niske `yes`. Ceo proces može da potraje, najviše u delu pravljenja Kubernetes klastera i skupa čvorova koji može da oduzme i do 20 minuta.

Ako ovaj rad čitate godinu dana ili više nakon njegovog pisanja, verovatno ćete dobiti grešku poput `400-InvalidParameter, Invalid kubernetesVersion: Invalid Kubernetes version` koja znači da verzija Kubernetesa iz primera više nije podržana (trenutna politika je da su podržane poslednje tri verzije, a nove verzije se objavljuju svaka 4 meseca, što OCI prati sa malim zakašnjenjem). Sve što je potrebno uraditi je ažurirati verziju (i, potencijalno, identifikator slike čvora koja će biti postavljena na mašini, `node_image_id`, što je i naznačeno u opisu promenljive) i pokrenuti `terraform apply` ponovo.

Ovaj put, Teraform zna da treba samo da napravi dva preostala resursa, Kubernetes i skup čvorova, što će i ponuditi da uradi. Takođe ćemo u planu videti i da su popunjene vrednosti nešto više argumenata koje su sada poznati, poput `compartment_id`. S obzirom na prirodu infrastrukture u oblaku, svih povezanih usluga i snabdevača koji su uključeni u proces, može doći i do nekih drugih grešaka koje su posledica zastarevanja nekog alata ili dela ponude snabdevača. Ovo je praktično neizbežno, ali srećom se najčešće lako rešava uz nekoliko pretraga po ključnim

rečima iz poruke o grešci. Na realnim projektima, treba nastojati da se svi alati i infrastruktura redovno održavaju i ažuriranja redovno prate uz čitanje relevantnih opisa svakog izdanja, što eliminiše velike grupe ažuriranja i daje nam vremena da zamenimo zastarele delove pre nego što istekne period podrške.

Ako je sve dobro prošlo, na kraju očekujemo da vidimo poruku koja počinje sa `Apply complete!` i listu izlaznih promenljivih, u ovom slučaju samo javnu adresu bastion instance. Na nju se možemo povezati pomoću komande `ssh opc@${IP_ADRESA}`, postaviti `kubectl` i upravljati Kubernetes klasterom, što je van opsega ovog rada (ali ćemo u trećoj glavi prikazati jedan drugačiji način upravljanja Kubernetesom). Takođe možemo da se ulogujemo na veb konzolu i uverimo se da su svi resursi napravljeni.

Ako opet pokrenemo `terraform apply`, videćemo izlaze koji nam kažu da Teraform proverava trenutno stanje i, ako nismo ništa menjali, poruku da nema izmena i da nema šta novo da se uradi. Ako pak dodamo neki novi resurs ili promenimo postojeći, Teraform će da izvrši akcije na infrastrukturi kako bi ispoštovao nova podešavanja. U slučaju menjanja resursa, u zavisnosti od toga šta promenimo, resurs se može direktno izmeniti (ako smo promenili argument koji u dokumentaciji resursa ima oznaku `(Updatable)`) ili obrisati i ponovo napraviti s novim podešavanjima. U oba slučaja, Teraform će nam reći šta planira da uradi i tražiti potvrdu.

Ako nam je ovo služilo samo kao svojevrstan test ili igra, sve napravljeno možemo uništiti jednom komandom: `terraform destroy`, koja pravi plan:

```
Plan: 0 to add, 0 to change, 15 to destroy.
```

Kroz desetak minuta, svi resursi će biti obrisani sa okruženja u oblaku. Ovo je *značajno* lakše nego ručno brisanje, pošto pre brisanja bilo kog većeg resursa (uključujući i *compartment*) OCI zahteva da se obrišu svi resursi koji od njega zavise, što je ručno izuzetno zamoran posao.

Ovime završavamo naš (mali) primer postavljanja Kubernetes infrastrukture na OCI oblaku pomoću Teraforma, u nadi da je čitaocu nešto jasnije zašto je infrastruktura kao kod primamljiv koncept i kako koristiti Teraform. U narednoj glavi ćemo se fokusirati na drugi alat i nešto izmenjene principa sa ambicioznijim ciljem, ali videćemo da će se isti resursi sa istim argumentima opet pojaviti, samo u drugačijoj sintaksi.

Glava 3

Programiranje infrastrukture

Pomoću Teraforma možemo opisati bilo kakvu infrastrukturu u oblaku, tako da se nameće pitanje zašto bismo tražili neku drugačiju alternativu? Zaista, u jednostavnijim slučajevima poput primera iz prethodne glave, nema nikakve potrebe i Teraform radi posao savršeno. Izložimo sad neke primere koji bi gurali Teraform do njegovih granica.

Nije teško zamisliti slučaj gde neki resurs treba napraviti samo u određenoj situaciji, ili ga napraviti u nekoj „petlji" (npr. za svakog korisnika, napraviti IAM nalog). Ovo je moguće opisati HCL-om, pomoću `count` i `for_each` meta-argumenata, po potrebi sa ternarnim operatorom, međutim već se nazire da ovo nije baš slučaj upotrebe za koji je HCL stvoren. Ako želimo da pročitamo neka podešavanja iz datoteke, možemo koristiti ugrađenu `file()` funkciju; ako je ta datoteka u YAML formatu i želimo da ga parsiramo, možemo koristiti ugrađenu `yamldecode()` funkciju. Ako je ta datoteka u nekom nestandardnom formatu, potrebno ju je obraditi na neki način, ili je npr. u pitanju neka kompaktna baza podataka, nemamo sreće, pošto Teraform ne podržava korisnički definisane funkcije već samo one koji su njegovi razvijači definisali.

Ako želimo da Teraform skriptu integrišemo u neku veću aplikaciju, ne postoji zgodan interfejs koji bi nam to olakšao. Ako imamo programersku pozadinu i navikli smo se na rad u nekom naprednom razvojnom okruženju, ono možda prepoznaje HCL, ali verovatno nije ni blizu toliko korisno kao za jezik za koji je primarno napravljeno. Ako želimo da koristimo funkcionalnost omiljene biblioteke ili deo logike koji smo prethodno napisali za drugu svrhu, to će biti ili nemoguće ili će implementacija te logike na jeziku koji Teraform razume zahtevati sličnu količinu napora kao njeno pisanje iznova. Ako hoćemo da pišemo testove, postoje specijalni blokovi koji

nam to omogućavaju, ali način na koji se to radi je dosta verbozan i u poređenju sa modernim okvirima za testiranje ograničen, što je i očekivano s obzirom da je u trenutnom obliku dostupan tek od oktobra 2023. godine [3].

Jedna od najprepoznatljivijih osobina Teraforma, HCL jezik, je takođe osobina koja ga najviše ograničava. Ekosistem dodatnih funkcionalnosti i alata nikada neće moći da se takmiči sa ekosistemima koji postoje oko popularnih konvencionalnih programskih jezika. Svojevrsno priznanje ove činjenice je projekat *Cloud Development Kit for Terraform* (CDKTF) iste kompanije koja razvija Teraform koji omogućava korišćenje poznatih programskih jezika (trenutno: TajpSkripta, Pajtona, Jave, C# i Goa) za definisanje infrastrukture, inspirisan i delimično baziran na sličnom Amazonovom projektu *AWS CDK*. Međutim, ovo je prilično nov napor i u trenutku pisanja ovog rada nije spreman za produkcionu upotrebu [6].

Praktično isti koncept, ali u nešto zrelijem i stabilnijem izdanju, je dostupan u okviru alata Pulumi čija prva stabilna verzija je objavljena u septembru 2019 [4]. Od programskih platformi i jezika podržava *Node.js* (Javaskript i TajpSkript), Pajton, Go, .NET (C#, F#, VB) i Javu, veliki broj snabdevača i modula ekvivalentnih Teraformu, a postoji i alat koji može prevesti bilo koji Teraform snabdevač otvorenog koda u odgovarajući Pulumi kod [13].

Osnovni Pulumi programi su i dalje u suštini deklarativni, s obzirom da se resursi definišu deklaracijama promenljivih, ali možemo koristiti standardne konstrukte za kontrolu toka, kao i sve alate za programski jezik na koje smo navikli. Međutim, u svakom trenutku možemo „iskočiti" iz ovog deklarativnog moda i izvršiti neku logiku ili pozvati funkciju, uz par napomena koje se tiču vremena izvršavanja te logike koje ćemo videti kroz projekat. Takođe, uz standardne okvire za testiranje, imamo opciju korišćenja i alata specijalizovanih za testiranje IaC programa kao što je ProTI [17].

3.1 Osnovni koncepti Pulumi alata i programa

Između Teraforma i Pulumija postoje određene sličnosti, pre svega u korišćenju i definisanju resursa. Pulumi resurse definiše kao klase (ili strukture u Go-u; u nastavku ćemo pretpostaviti da radimo sa objektno-orijentisanim jezikom) čiji konstruktor prima odgovarajuće argumente i njihovim instanciranjem se pravi resurs. Nazivi ovih klasa i argumenata najčešće odgovaraju nazivima u Teraformu. Ekviva-

lent Teraform modulima su komponente koje su ništa više do klase koje programer[1] definiše i koje proširuju odgovarajuću klasu iz Pulumi paketa.

Snabdevač u oba alata ima isto značenje, a u Pulumi programima je predstavljen *dependency*-jem koji se u program uključuje kao i svaka druga biblioteka i može se dodatno podesiti pravljenjem instance odgovarajuće klase iz uključenog paketa[2]. Isto važi i za koncepte stanja i *storage backend*-a koji imaju isto značenje u oba alata. Još jedna sličnost Pulumija i Teraforma je upotreba istoimenih CLI alata, mada ćemo videti da se Pulumi programi mogu pokretati i na druge načine.

Ulazne i izlazne promenljive predstavljaju isti koncept na oba mesta, sa vrlo prirodnom upotrebom u Pulumi programima (deluju kao i sve druge promenljive), međutim sa nešto kompleksnijom implementacijom. Svi izlazi su zapravo instance (generičkog) tipa *Output* koji je nalik obećanju (*promise*): pošto su resursi samo deklaracije koje se „izvršavaju" odmah, očekujemo da oni u trenutku evaluacije nemaju vrednost, već će je dobiti u nekom budućem trenutku[3]. Ulazne promenljive resura i komponenti najčešće primaju neki konkretan tip, ali i odgovarajući *Output* tip, kako programer ne bi morao posebnu pažnju da posvećuje ovom implementacionom detalju. Ako bismo želeli da zapravo pristupimo vrednosti izlazne promenljive za potrebe neke naše logike, Pulumi nam na raspolaganje stavlja funkcije `apply` (za jedan izlaz) i `all` (za niz izlaza) koje primaju lambdu čiji argument je vrednost izlaza i koja se izvršava u onom trenutku kada izlazna promenljiva dobije vrednost.

Da bismo neki Pulumi program mogli da izvršimo, on mora biti deo projekta, tj. mora postojati `Pulumi.yaml` datoteka koja minimalno definiše ime, opis i programsku platformu (postoje i razni opcioni parametri, poput podešavanja *storage backend*-a, koji su popisani u dokumentaciji: `https://www.pulumi.com/docs/concepts/projects/project-file/`). Svaki Pulumi program se pušta u rad na nekom *steku* koji predstavlja skup podesivih opcija i za koji se vezuje stanje. Na primer, možemo imati odvojene stekove za razvojno i za produkciono okruženje sa nezavisnim podešavanjima i stanjima. Pulumi projekat zasnovan na nekom šablonu i jedan stek možemo napraviti komandom `pulumi new` i praćenjem instrukcija.

[1] Osobu koja piše Pulumi programe ćemo nazivati programer, iako je možda na poziciji DevOpsa.

[2] S obzirom da su snabdevači takođe klase koje proširuju neku drugu klasu, pravljenje novog snabdevača sa nekim skupom funkcionalnosti koji nam nedostaje je takođe značajno pojednostavljeno. Ovo nećemo detaljnije pokriti u ovom radu, ali zainteresovani čitalac može pronaći više informacija na: `https://www.pulumi.com/docs/concepts/resources/dynamic-providers/`

[3] Kada tačno izlazi dobijaju vrednost zavisi od logike snabdevača. Pred kraj projekta ćemo videti primer resursa gde se izlazi odmah popunjavaju i praktične implikacije toga.

Podešavanja steka (eng. *configuration*) se predstavljaju YAML datotekom `Pulumi.<naziv_steka>.yaml` u koju možemo dodavati proizvoljne podatke koje ćemo kasnije čitati iz programa. Ovu datoteku možemo menjati i pomoću komande `pulumi config set`. U ovoj datoteci možemo definisati i poverljive podatke: ako prethodnoj komandi prosledimo opciju `--secret`, ona će podatke šifrovati i posebno označiti u datoteci. Ako se ključ za šifrovanje ne nalazi u predefinisanoj promenljivoj okruženja (*environment varijabli*), CLI će nas upitati da ga unesemo, kako pri postavljanju ovih podataka tako i svaki put kada budemo izvršavali program.

3.2 Struktura i postavka projekta

Primer iz prethodna glave je služio da demonstrira upotrebu Teraforma i čitaoca uvede u svet infrastrukture kao koda. U ovoj glavi, opisaćemo nešto složeniji projekat. Želimo da napišemo program koji će na osnovu datoteke u kojoj se nalazi spisak timova, njihovih projekata i korisničkih imena članova na GitHubu, inicijalizovati repozitorijume na GitHubu na osnovu nekog šablonskog repozitorijuma, postaviti Kubernetes infrastrukturu na Oraklovom oblaku i podesiti sve prateće alate tako da se projekti automatski izgrade i isporuče na infrastrutkru u oblaku i postanu javno dostupni na poddomenu koji odgovara nazivu projekta. Svaki *commit* na glavnu granu bi takođe trebalo da okine proces izgradnje i isporučivanja projekata, bez ikakve intervencije programera. Ovakvi sistemi sigurno postoje u većim korporacijama, ali pokazaćemo da ih je moguće implementirati i kao Pulumi program(e) i diskutovati prednosti i mane ovog pristupa.

Jedna od prvih odluka koju moramo napraviti kada koristimo Pulumi je odabir programskog jezika. Javu i .NET jezike ima smisla koristiti ako posedujemo ekspertizu u tim oblastima i želimo da naš program integrišemo u veći projekat koji je deo nekog od ovih sistema, ali ovi jezici nisu preterano popularni u DevOps zajednici i mogu biti previše verbozni. Go možda deluje privlačno, ali s obzirom da su generički tipovi relativno novi dodatak jeziku, Pulumijeva podrška za ovaj jezik je podeljena na „novi" (sa genericima) i „stari" (bez generika) Go i za „novi" postoji daleko manje dokumentacije i podrške zajednice, što čini ovaj izbor manje poželjnim. Ostaje izbor između TajpSkripta[4] i Pajtona, gde ne možemo previše pogrešiti. Mi smo za

[4] JavaSkript nema nijednu objektivnu prednost u odnosu na TajpSkript u ovom slučaju i nećemo ga razmatrati.

ovaj projekat odabrali da radimo u TajpSkriptu s obzirom na nešto moćniji tipovni sistem.

Izvorni kod ovog projekta dostupan je na GitHubu na adresi: `https://github.com/luka-j/pulumi-oci-k8s`. Primetićemo da je ovo *Node.js* program, a ne Pulumi projekat; nešto više o ovome ćemo pričati u poslednjem poglavlju ove glave. U direktorijumu `stacks/`, naći ćemo četiri poddirektorijuma, svaki sa po jednim Pulumi projektom, koje ćemo opisati u narednim poglavljima. Ovakva podela na više Pulumi projekata doprinosi modularnosti i omogućava upotrebu samo nekog podskupa funkcionalnosti u zavisnosti od potreba, kao i njihovo lakše uključivanje u neki drugi veći sistem.

Za pokretanje programa, potrebno je da imamo postavljen *Node.js*, a za pokretanje pojedinačnih stekova i Pulumi CLI. U ovom projektu nećemo posebno podešavati OCI snabdevača (iako je to moguće, na način analogan Teraformu), već ćemo se osloniti na Pulumi da prepozna podešavanja koja koristi i Oraklov CLI alat, a čija uputstva za postavljanje se nalaze na sledećoj adresi: `https://docs.oracle.com/en-us/iaas/Content/API/SDKDocs/cliinstall.htm`.

3.3 OCI infrastrutkura u Pulumiju

Infrastrukturu na Oraklovom klaudu opisujemo Pulumi programom koji se nalazi u direktorijumu `stacks/oci/`. U poddirektorijumu `components/` naći ćemo komponente koje ćemo kasnije spojiti, kao što smo to radili sa modulima u Teraformu. Ovo je još jedan stepen modularnosti, jer iste komponente možemo koristiti i u nekom drugom Pulumi programu ako za tim nastane potreba.

U ovom folderu se nalazi `Pulumi.yaml` datoteka koja označava da je ovo Pulumi projekat, kao i podešavanja u `Pulumi.oci_dev.yaml` datoteci (stekove različitih programa ćemo nazivati različito, kako bismo izbegli probleme kada ih budemo zajedno izvršavali na kraju). Datoteku sa podešavanjima možemo menjati direktno ili pomoću `pulumi config set` komande. Ostale datoteke čine standardnu postavku *Node.js* projekta pisanog u TajpSkriptu. Prvo ćemo obrazložiti komponente, a zatim ćemo proći kroz `index.ts` datoteku u kojoj ćemo ih sve spojiti.

VCN

Komponenta koja podešava mrežu je definisana u datoteci `vcn.ts`. Kako bi Pulumi prepoznao komponentu, potrebno je napisati klasu koja proširuje klasu `ComponentResource` iz paketa `@pulumi/pulumi` i implementirati konstruktor. Konstruktor svakog resursa, pa samim tim i komponente, prima tri argumenta: naziv resursa, objekat sa ulaznim promenljivama `args` i objekat koji predstavlja neke opcije koje se mogu odnositi na sve resurse (slično meta-argumentima u Teraformu). Ovakav konstruktor i mi definišemo u komponenti, a konstruktoru superklase uz ove argumente prosleđujemo i jedinstven naziv komponente koji će se interno koristiti. Što se tipova ulaznih promeljiva tiče, koristimo konkretan tip ili `Output` u zavisnosti od potreba pozivaoca. Nešto fleksibilniji (ali verbozniji) način bi bio deklarisati tip ulaznih promenljiva kao `tip | Output<tip>` ili `Input<tip>` (koji uz ova dva dozvoljava i `Promise<tip>`) svuda.

Razlika *VCN* modula u Teraform primeru i ove komponente je to što ovde nećemo koristiti gotov modul koji nam je prethodno skratio kod. Ovakav modul ne postoji za Pulumi, a iako ga je moguće „konvertovati", on koristi neke funkcionalnosti Teraforma koje alat za konverziju ne podržava i čak i ako to ručno popravimo morali bismo da taj kod uključimo i distribuiramo kao deo ovog projekta. S obzirom da postavljanje mreže za naše potrebe koristeći samo resurse koje imamo na raspolaganju nije toliko komplikovano, jednostavnije je ne uvoditi dodatnu kompleksnost kroz ovu zavisnost. Prvo ćemo napraviti resurs tipa `oci.core.Vcn` (iz `@pulumi/oci` paketa, kao i svi ostali koji se odnose na OCI snabdevača), koji je ekvivalent Teraformovom resursu `oci_core_vcn`:

```
this.vcn = new oci.core.Vcn(`${name}_vcn`, {
    compartmentId: args.compartmentId,
    displayName: args.vcnName,
    dnsLabel: args.vcnName,
    cidrBlocks: [args.vcnRange],
}, { parent: this })
```

S obzirom da smo videli slične resurse u Teraformu, sa znanjem osnova TajpSkripta (ili čak samo JavaSkripta), nije teško zaključiti šta se ovde dešava: ime resursa pravimo na osnovu imena komponente, kao ulazne promenljive prosleđujemo odgovarajuće ulazne promenljive komponente i u opcijama kažemo da je roditelj ovog resursa naša komponenta (`this`). Samo ovaj poslednji deo nismo videli ranije i

on kaže Pulumiju da kada ispisuje plan ovaj resurs smatra „detetom" komponente i, važnije, da ona treba da nasledi većinu opcija koje važe za komponentu, uključujući i snabdevača (nešto kasnije ćemo videti slučaj kada je ovo bitno).

Podmreže i sigurnosne liste (`security list`) smo videli u Teraform primeru i nećemo ih posebno komentarisati. Nazivi resursa i argumenata su isti kao i u Teraformu, samo koriste različitu konvenciju imenovanja, a argumentima koji su u Teraformu ugnježđeni blokovi odgovaraju obični objekti[5].

Ostaje još da napravimo tabele rutiranja i mrežne *gateway*-e koji su bili deo gotovog modula koji smo koristili u Teraform primeru. Za javnu podmrežu, pravimo instancu `oci.core.InternetGateway` i `oci.core.RouteTable` sa jednim pravilom koji sav saobraćaj rutira na `InternetGateway`. U ovoj podmreži će svi imati javnu IP adresu pa nema nekih komplikacija. Za privatnu podmrežu, pravimo instancu `oci.core.NatGateway` i `oci.core.RouteTable` sa jednim pravilom koji sav saobraćaj rutira na njega. U ovoj podmreži niko nema javnu IP adresu, ali želimo da imamo izlazni saobraćaj na internet, tako da će nam `NatGateway` služiti kao NAT sloj koji će sve u ovoj podmreži „sakriti" iza jedne IP adrese.

Kada inicijalizujemo sve promenljive, ostaje još da pozovemo `registerOutputs` metodu iz natklase kako bismo Pulumiju rekli da smo završili sa inicijalizacijom komponente i u retkim slučajevima kažemo šta su izlazne promenljive. Ona kao argument prima objekat koji opisuje izlazne promenljive, ali u principu bismo joj mogli proslediti i prazan objekat s obizrom da su naše komponente logička grupisanja resursa, od kojih svakako svaki ima svoje izlaze. Ovo nije najbolji primer dizajna i čak i neki od programera koji održavaju Pulumi imaju nedoumica [5][11], ali srećom u praksi nema značajne posledice i mi ćemo ga koristiti kao hint šta očekujemo da će se koristiti spolja. Na kraju, dodajemo `export` za ovu klasu kako bismo joj mogli pristupiti i iz drugih datoteka.

Kubernetes

Drugu komponentu koju ćemo ukratko opisati ovde smo takođe videli u Teraform primeru: Kubernetes klaster. U okviru nje, pravimo resurse `oci.containerengine.Cluster` i `oci.containerengine.NodePool` na potpuno analogan način kao ranije, sa praktično istim argumentima. Umesto virtuelne mašine koja nam je predstavljala bastion host, koristićemo resurs `oci.bastion.Bastion` koji ima sličnu svrhu. Za potrebe

[5]Argumentu tipa `object` bi odgovarao TajpSkript objekat tipa `{[key: string]: any}`, međutim takav primer ovde nećemo imati.

ovog projekta ne želimo da se ručno logujemo na udaljenu mašinu, već ćemo napraviti SSH tunel kroz Bastion instancu, pa nam upotreba ovog resursa umesto neke mašine pojednostavljuje podešavanja. Za kraj, u konstruktoru pozivamo `registerOutputs`, na kraju datoteke dodajemo direktivu `export default` i gotovi smo sa postavkom Kubernetes klastera.

Budžet

U ovom projektu imamo i jednu komponentu koju nismo koristili u Teraform primeru, `Budget`. Poenta budžeta je da ograniči potrošnju nekog konkretnog resursa ili neke grupe resursa. Budžet ima neku brojčanu vrednost u valuti naloga, a možemo definisati *pravila* kojima bliže određujemo u kojim slučajevima će nam stići upozorenje na mejl. Rekli smo da nam je cilj da koristimo samo besplatne resurse, ali za neke resurse to nije moguće uvek garantovati. Jedan primer takvog resursa je mrežni protok, pošto je to nešto što naše aplikacije koriste: na raspolaganju imamo 10TB izlaznog saobraćaja (*egress*) i Orakl može da nas upozori o potrošnji, ali upravljanje (i ubijanje) aplikacija je naša odgovornost.

Komponenta `Budget` se sastoji od dva tipa resursa, `oci.budget.Budget` i `oci.budget.Rule`. Prvi je vezan za *tenancy*, tako da tu prosleđujemo taj identifikator kao `compartmentId`[6], kažemo da je budžet mesečni i da se odnosi na odeljak čiji identifikator dobijamo kao ulaznu promenljivu. Kod definisanja pravila po prvi put vidimo jedan način možemo napraviti niz resursa:

```
this.rules = args.rules.map((rule, i) =>
    new Rule(`${name}_budget_rule_${i}`, {
        budgetId: this.budget.id,
        threshold: rule.threshold,
        thresholdType: rule.thresholdType,
        type: rule.type,
        description: rule.description,
        displayName: rule.displayName,
        message: rule.message,
    }, { parent: this }));
```

[6]Trenutno, dokumentacija sadrži šablonski tekst koji sugeriše da se ovde zaista traži identifikator *compartment*-a, međutim poučeni prethodnim iskustvom znamo da resursi koji nisu deo *compartment*-a zapravo traže identifikator *tenancy*-ja.

U `args.rules` nam se nalazi niz objekata koji opisuju pravila, a čiji tip smo definisali na standardan način nešto više u istoj datoteci. Kao i nad svim nizovima, i nad ovim možemo primeniti ugrađenu funkciju `.map()` i napraviti instance `rule`-ova. Ovakav niz na kraju dodeljujemo instancnoj promenljivoj, kako bi mogli da joj pristupimo i iz drugih datoteka koji vide ovu klasu. Ovime smo definisali sve resurse ove komponente, a njihov tačan broj će zavisiti od konkretnih argumenata koji joj se proslede, što Pulumiju ne predstavlja problem.

Dodatni resursi i instanciranje komponenti

Spajanje ovih komponenti se vrši u `index.ts` datoteci. Već na samom vrhu datoteke vidimo relevantne direktive za uvoz, koje se nikako ne razlikuju od standardnih TajpSkript programa:

```
import Vcn from "./components/vcn";
import KubernetesCluster from "./components/cluster";
import Budget from "./components/budget";
```

Ispod importa instanciramo objekat tipa `pulumi.Config`: `const config = new Config();` koji ćemo koristiti za čitanje podešavanja steka. Podrazumevano se koristi onaj stek sa kojim je pokrenut program.

U ovom projektu nismo definisali *compartment* kao deo neke komponente, pa ćemo u `index.ts` napraviti odgovarajući resurs:

```
const compartment = new oci.identity.Compartment("master", {
    name: config.require("compartmentName"),
    description: config.require("compartmentDescription"),
    enableDelete: true
});
```

Argument `enableDelete` je jedna od retkih specifičnosti Pulumi snabdevača i ne postoji u Teraform snabdevaču; kada se postavi na *true*, snabdevač će izbaciti grešku ako odeljak sa istim imenom već postoji i obrisaće ga ako se ukloni njegova definicija iz koda. Podrazumevana vrednost, *false*, je odgovarajuća ako želimo da koristimo postojeći odeljak umesto da forsiramo pravljenje novog.

Druga stvar koju smo imali u *Compartment* modulu u Teraform primeru je lista *availability domena*, koja će nam i ovde trebati. Ovo je u Teraformu bilo predstavljeno *data* blokom, koji su u Pulumiju konvencionalno funkcije čiji nazivi počinju za *get*.

U ovom slučaju, želimo da koristimo funkciju `oci.identity.getAvailabilityDomains` koja kao argument prima objekat koji sadrži polje `compartmentId` tipa *string*. Ako pokušamo da samo prosledimo `compartment.id`, dobićemo grešku da se tipovi ne poklapaju, pošto je prosleđena promenljiva tipa *Output<string>*. Ovde moramo koristiti nešto ranije pomenutu `apply` metodu, pa će ovaj blok izgledati ovako:

```
const ads = compartment.id.apply(id =>
    oci.identity.getAvailabilityDomains({
        compartmentId: id
    })
);
```

Ovime smo definisali promenljivu `ads` tipa *Output*. Ovaj deo koda će se izvršiti odmah, ali će *Output* zapravo dobiti smislenu vrednost tek kada se *compartment* napravi i izvrši pozvana metoda za dohvatanje zona. Ovo nas ni na koji način ne ometa da koristimo promenljivu i prosleđujemo je drugim resursima koji na ulazu očekuju *Output*-e.

Ostali resursi se prave po istom šablonu kao i do sad, nezavisno od toga da li ih definiše snabdevač ili su zapravo komponente. Njihove ulazne promenljive su ili konstante, ili izlazi drugih resursa, ili parametri iz podešavanja. U poslednjem slučaju, koristimo `config` objekat kako bismo pročitali neki konkretan parametar, i to metodu `require()` ako je u pitanju podatak koji nije poverljiv (postavljen je bez upotrebe `-secret` opcije) ili `requireSecret()` ako jeste. Ako je neki parametar opcion, možemo koristiti `get()` i `getSecret()` metode koje će vratiti `undefined` u slučaju da parameter ne postoji, za razliku od `require()` metoda koje će izbaciti grešku. Sve metode imaju i analogno nazvane metode u slučaju da je parametar logička promenljiva, broj ili objekat, pa tako možemo pozvati i `requireBool()`, `requireNumber()` ili `requireObject()` u skladu sa potrebama. Jedina praktična razlika u upotrebi metoda za dohvatanje poverljivih podataka je što one vraćaju objekte tipa *Output*, dok ove druge vraćaju *string*, *bool*, *number* ili neki objektni tip.

Ostaje još jedan resurs koji nismo imali u Teraform primeru, a vezan je za bastion koji smo napravili kao deo Kubernetes komponente. Da bismo mogli da se povežemo na njega, potrebna nam je *sesija*, koju opisujemo resursom `oci.bastion.Session`. Njoj treba da prosledimo naš javni SSH ključ i podatke o ciljnom resursu. Ako bastion posmatramo kao tunel, naša mašina je jedan kraj tunela, a ciljni resurs je

drugi, tj. mi se na ciljni resurs povezujemo kroz bastion (preciznije, pakete namenjene ciljnom resursu šaljemo bastionu, koji ih dalje prosleđuje). U našem slučaju, ciljni resurs je Kubernetesov endpoint koji dobijamo sledećom linijom:

```
const clusterPrivateEndpointParts =
            cluster.okeCluster.endpoints[0].privateEndpoint
                    .apply((t) => t.split(":"));
```

Promenljiva `cluster` sadrži našu komponentu, `okeCluster` je njeno polje koje predstavlja OCI resurs, a `endpoints[0].privateEndpoint` jedan od njenih izlaza koji je upravo Kubernetes endpoint u formatu `IP_ADRESA:PORT`. Mi ovu nisku delimo na dve kod dvotačke, pa dobijamo promenljivu koja je tipa *Output<string[]>*. Nju ćemo dalje koristiti u definiciji bastion sesije:

```
const bastionSession = new oci.bastion.Session("bastion_session", {
    bastionId: cluster.bastion.id,
    keyDetails: {
        publicKeyContent: config.require("publicSshKey")
    },
    targetResourceDetails: {
        sessionType: "PORT_FORWARDING",
        targetResourcePrivateIpAddress:
                clusterPrivateEndpointParts.apply((p) => p[0]),
        targetResourcePort:
                clusterPrivateEndpointParts.apply((p) => parseInt(p[1])),
    },
    sessionTtlInSeconds: 60*60,
}, {
    deletedWith: cluster.bastion
});
```

Kod prva dva argumenta u ulazu nema novina. `targetResourceDetails` argumentom definišemo ciljni resurs i ovde kažemo da ćemo ga koristiti za prosleđivanje sadržaja kroz neki port (*port forwarding*). Kako je `clusterPrivateEndpointParts` tipa *Output*, da bismo pristupili nekom elementu ovog niza, opet nam je potrebna *apply* funkcija. Pošto su ove sesije kratkotrajne, argumentom `sessionTtlInSeconds` određujemo njihov životni vek, u ovom slučaju na sat vremena. Ovu sesiju ćemo kasnije koristiti da napravimo SSH tunel između naše mašine i Kubernetesa gde ćemo

uspostaviti SSH vezu sa sesijom, tako da kad pošaljemo paket na `localhost:6443`, on će zapravo biti poslat na bastion, koji će ga zatim proslediti na Kubernetes endpoint, što će nam omogućiti upravljanje privatnim Kubernetes klasterom sa lokalne mašine.

U ovom resursu možemo videti i jednu novu opciju, `deleteWith`, koja snabdevaču kaže da ako se briše i resurs koji ovde prosleđujemo, nema potrebe brisati i resurs na kog se ova opcija odnosi. Konkretno, ako brišemo bastion, nema potrebe brisati i sesiju. Iako na prvi pogled deluje samo kao „lep dodatak", ovo ima i jednu mnogo suptilniju svrhu, a to je omogućavanje brisanja steka (ili makar dela koji uključuje bastion) nekad u budućnosti. Kada životni vek sesije istekne, ona će se automatski obrisati i njeno *ponovno* brisanje neće biti moguće. Mi možemo sinhronizovati realno stanje sa onime što je Pulumi zapamtio pomoću `pulumi refresh` komande, ali ako to ne uradimo u ovom slučaju ćemo dobiti grešku kada Pulumi pokuša da obriše sesiju ako ona ne postoji. Upotrebom `deletedWith`, praktično kažemo Pulumiju da preskoči brisanje ovog resursa ako briše i bastion, čime zaobilazimo ovaj problem, a sesija će se svakako implicitno obrisati (ako postoji) kada se obriše bastion.

Zbog svog roka trajanja, bastion sesije su jedan od retkih resursa kojima zapravo nije toliko pogodno upravljati pomoću alata za definisanje infrastrukture kodom. One nakon određenog vremena same od sebe menjaju stanje (iz *postoji* u *ne postoji*), što se kosi sa osnovnim principima infrastrukture kao koda. Ovde ih koristimo kako bismo izbegli ljudsku interakciju u celom procesu, ali u svakodnevnom radu, sasvim je opravdano (i mnogo zgodnije) napraviti ih kroz grafičku konzolu ili CLI alate.

Na kraju datoteke imamo `export` direktivu. Sve što se `export`-uje iz steka, smatra se izlaznom promenljivom i ispisuje se na standardni izlaz nakon uspešne primene izmena. Ovome ćemo takođe moći da pristupimo na programatički način, ako je stek deo većeg programa, što ćemo videti na kraju ove glave.

Ako se pozicioniramo u direktorijum `stacks/oci`, napravimo potrebne izmene podešavanja sa `pulumi config set` (npr. parametra `tenancyId`, ili Kubernetes verzije i slika), izvršavanjem `pulumi up` komande bi trebalo da dobijemo plan izvršavanja koji opisuje šta će sve biti napravljeno od infrastrukture. Očekujemo da na kraju ovog plana vidimo linije:

```
Resources:
    + 21 to create
```

Potvrdom plana, Pulumi kreće u akciju. Slično kao u Teraform primeru, nakon 10-20

minuta imaćemo spreman Kubernetes klaster na OCI infrastrukturi.

3.4 Generalizacija IaC pristupa na druge servise

U prošloj glavi, ovde smo završili sadržaj primera i posavetovali čitaoca da se poveže na klaster i sam smisli kako da ga podesi, međutim sada želimo da idemo korak dalje. Spomenuli smo da se alati za opisivanje infrastrukture kodom ne ograničavaju samo na snabdevače okruženja u oblaku, već je na taj način moguće podesiti i druge stvari. Jedan od popularnijih Pulumi (a i Teraform) snabdevača koji nije vezan za snabdevače infrastrukture u oblaku je *Kubernetes*, koji nam omogućava upravljanje Kubernetes klasterom pomoću omiljenog IaC alata[7]. U okviru ovog poglavlja ćemo takođe videti i *GitHub* i *Cloudflare* snabdevače, ali ćemo njima posvetiti nešto manje vremena.

Najočiglednija prednost korišćenja ovih alata i za opisivanje stvari koji nisu infrastruktura u oblaku je pojednostavljena integracija sa delom koji jeste infrastruktura u oblaku. Međutim, mi smo već razdvojili ove stekove, tako da smo mogli i da se umesto pisanja Pulumi programa integrišemo sa odgovarajućim API-jem. Ovo bi značilo čitanje dokumentacije za tri (dodatne) različite platforme i njihove API-je koji nisu međusobno konzistentni i imaju svoje specifičnosti u svakom delu. Iako možemo reći da svaki Pulumi snabdevač ima svoje specifičnosti, oni imaju mnogo manje „stepeni slobode" i značajno je jednostavnije pisati sve integracije u istom maniru. Tu su takođe i druge prednosti Pulumija, pre svega upravljanje podešavanjima i stanjem, koje nam omogućava da ažuriramo program i logiku migracija prepustimo alatu, umesto da pišemo gomilu skripti koje bi usklađivale trenutno i očekivano stanje.

Kubernetes

Krajnji cilj projekta je da pokrene neke aplikacije u Kubernetes klasteru i da one budu javno izložene. Da bi ovo postalo stvarnost, treba nam par alata unutar klastera. Prvo ćemo postaviti *ingress-nginx*, upravljač ulaznog saobraćaja baziran na *nginx* projektu koji održava Kubernetes tim (ne treba ga mešati sa *nginx-ingress*, drugim upravljačem baziranom na *nginx*-u koji održava *nginx* tim). Zatim ćemo po-

[7]Postoje ljudi koji veruju da ovaj odnos treba da bude obrnut, tj. da se pomoću Kubernetes objekata treba opisivati infrastruktura u oblaku. Trenutno najpopularniji alat ovog tipa je *Crossplane*, dostupan na `https://www.crossplane.io/`.

staviti *cert-manager* koji će za nas dobavljati i podesiti TLS sertifikate. Na kraju, postavićemo *ArgoCD* alat koji će pokretati druge aplikacije u klasteru na osnovu podešavanja koja ćemo kasnije postaviti na GitHub. Kao i u prethodnom poglavlju, ovo će biti stek za sebe u `stacks/k8s` direktorijumu sa svim pratećim podešavanjima, a komponente ćemo naći u `components/` poddirektorijumu.

Najjednostavniji način da postavimo mnoge aplikacije u Kubernetes klaster je pomoću alata *Helm*. On je praktično upravljač paketa (*package manager*) za Kubernetes. Helm ove „pakete" naziva *chart*-ovima, a instancu *chart*-a (kada se postavi) *release*. Ovaj resurs je zapravo skup drugih Kubernetes objekata koji omogućava da aplikacija radi nesmetano, uključujući *Deployment*-e, *Service*-e, *ConfigMap*-e i mnoge druge. Svaki *chart* ima podešavanja koje se po konvenciji čitaju iz *values.yaml* datoteke. Dokumentaciju *chart*-a, uključujući i spisak mogućih vrednosti koje možemo podesiti, ćemo naći na zvaničnoj stranici alata koji želimo da dodamo ili (čak i češće) na `https://artifacthub.io/` pomoću kog možemo pronaći pakete i podešavanja za druge alate, ali predstavlja i standardni pretraživač za Helm resurse. Konvencionalno, Helm ima svoj CLI alat pomoću kog postavljamo *chart*-ove, ali Kubernetes snabdevač u Pulumiju zna da postavi *chart-ove* bez ikakvih posebnih podešavanja.

Upravljač ulaznog saobraćaja

U datoteci `ingressController.ts` se nalazi definicija Helm *release*-a koji postavlja *ingress-nginx* i opisan je Pulumi resursom `kubernetes.helm.v3.release.Release`. Podatke o *chart*-u (`chart`, `version`, `repository`) možemo videti na odgovarajućoj *Artifact Hub* stranici: `https://artifacthub.io/packages/helm/ingress-nginx/ingress-nginx` i kopirati ih u definiciju resursa. Nazive imenskog prostora i *release*-a mi postavljamo i argumentom `createNamespace` kažemo Helmu da napravi imenski prostor za nas, s obzirom da u trenutku instanciranja resursa neće postojati. Ostaje još da popunimo `values`, koji je ovde objekat koji odgovara strukturi *values.yaml* datoteke odgovarajućeg *chart*-a.

Svi parametri koji su nam zanimljivi se nalaze u `controller` objektu, i to `replicaCount` da definiše koliko podova treba podići inicijalno, `resources` da definiše koliko resursa zahtevaju i `autoscaling` da odredi pod kojim uslovima i u kojoj meri Kubernetes treba da poveća broj podova u zavisnosti od iskorišćenosti traženih resursa. Objektom `service` podešavamo Kubernetes objekat tipa *Service* koji će rutirati saobraćaj na upravljač i ovde postavljamo anotaciju:

```
"oci.oraclecloud.com/load-balancer-type": "nlb"
```

Ona kaže Oraklu da hoćemo da se za ovaj servis digne *Network Load Balancer* (stoga `nlb`), umesto podrazumevanog *Load Balancer*-a. Ovaj prvi je L3/L4 raspoređivač opterećenja (za IP protokol radi na transportnom sloju, inače na mrežnom; ne podržava napredna pravila za rutiranje), dok je podrazumevani L7 raspoređivač (radi na aplikativnom sloju); nama je od Orakla dovoljno L4 rutiranje, s obzirom da će upravljač ulaznog saobraćaja da se pobrine za ostalo. Na kraju, omogućavamo izvoz metrika koje posle možemo obrađivati alatima poput *Prometheus*-a i *Grafane*.

cert-manager

Drugi alat koji želimo da podignemo je *cert-manager* koji će se brinuti o TLS sertifikatima za našu infrastrukturu. Ovo možda ne deluje kao toliko kritična briga, međutim u današnje vreme se maltene podrazumeva da svaki veb servis podržava HTTPS, a kao što ćemo videti postavka istog je jednostavna i ne košta ništa. Njegovi resursi su definisani u `certmanager.ts` datoteci. Kao i kada smo postavljali upravljač ulaznog saobraćaja, i ovde ćemo koristiti odgovarajući *Helm chart*, ali sa mnogo jednostavnijim `values` objektom:

```
values: {
    installCRDs: true
},
```

`CRD` je ovde (a i u Kubernetesu generalno) skraćenica za *Custom Resource Definition*, tip objekta koji definiše druge tipove objekta. Ovim parametrom kažemo da želimo da se i ove definicije tipova postave (instaliraju) zajedno sa *release*-om[8].

Jedan od CRD-ova koji *cert-manager* dodaje je *ClusterIssuer* koji opisuje izdavaoca sertifikata, tj. kaže *cert-manager*-u kako da dođe do TLS sertifikata. Pulumi naravno ne zna unapred koje mi sve CRD-ove možemo da dodamo i nema posebne tipove za svaki, ali ima resurs tipa `CustomResource` čija struktura odgovara YAML reprezentaciji Kubernetes objekata, pa tako i može da opiše bilo šta što želimo imati u našem klasteru. Napravićemo jedan `CustomResource` koji opisuje *ClusterIssuer*-a:

```
this.clusterIssuer = new CustomResource(`${name}_cluster_issuer`, {
    apiVersion: "cert-manager.io/v1",
```

[8]Ovo je posebna opcija u `cert-manager`-u zbog njegovog specifičnog odnosa sa CRD-ovima; detaljnije obrazloženje se može pronaći na zvaničnoj stranici: `https://cert-manager.io/docs/installation/helm/#crd-considerations`.

```
    kind: "ClusterIssuer",
    metadata: { ... },
    spec: {
        acme: {
            server: "https://acme-v02.api.letsencrypt.org/directory",
            email: args.acmeEmail,
            privateKeySecretRef: {
                name: "letsencrypt-prod"
            },
            solvers: [{
                http01: {
                    ingress: {
                        "class": "nginx"
                    }
                }
            }]
        }
    }
}, { parent: this, dependsOn: this.certManager });
```

Argumentima `apiVersion` i `kind` određujemo tip objekta. U specifikaciji (`spec` objekat) naziv jedinog objekta predstavlja tip izdavača sertifikata[9] i on sadrži relavantna podešavanja. Koristićemo *Let's Encrypt* CA, sertifikat ćemo da smestimo u *Secret* `letsencrypt-prod`, a kako bismo dokazali da mi imamo kontrolu nad nekim domenom (koji će biti definisan u *Ingress* resursu) koristićemo *solver* koji da koristi upravljač klase `nginx` da definiše odgovarajuću putanju koju će CA da proveri. Više detalja o ovoj metodi provere, tzv. *HTTP01 challenge*-u, može se naći u *Let's Encrypt* dokumentaciji na adresi: `https://letsencrypt.org/docs/challenge-types/#http-01-challenge`. Na osnovu ovih podešavanja *cert-manager* će zahtevati i postaviti nove sertifikate po potrebi svaki put kada se u klaster doda objekat tipa *Ingress* sa anotacijom `cert-manager.io/cluster-issuer` čija vrednost je ista kao i naziv ovog *ClusterIssuer*-a[10].

[9] `acme` je ovde skraćenica za *Automatic Certificate Management Environment* protokol, a ne naziv korporacije čije proizvode Pera Kojot koristi u pokušajima da ulovi Pticu Trkačicu.

[10] Konkretan mehanizam koji ovo omogućava se naziva *admission webhook* i dokumentovan je na sledećoj adresi: `https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/`.

Primetimo da ovde u opcijama sem `parent` polja imamo i `dependsOn`. Pošto `CustomResource` nigde ne referencira `Release` koji je prethodno napravljen, Pulumi ne zna za zavisnost između ova dva i podrazumevano će pokušati da ih istovremeno napravi. Ovo će dovesti do greške, pošto CRD koji definiše ovaj objekat (skoro sigurno) nije još primenjen. Specifikovanjem `dependsOn` kažemo Pulumiju da sačeka da se resurs `this.certManager` napravi pre nego što počne da pravi `ClusterIssuer`-a.

ArgoCD

Ostaje da dodamo još jednu komponentu pre nego što naš klaster možemo nazvati u potpunosti funkcionalnim, pošto i dalje nemamo način da puštamo aplikacije u rad na klasteru na neki „lep" način (bez upotrebe CLI alata). Ovaj problem će nam rešiti *ArgoCD*, alat za neprekidnu isporuku (*continuous delivery*) za Kubernetes. U suštini, Argo posmatra određeni direktorijum u nekom objavljenom (javnom ili privatnom) Git repozitorijumu i pravi Kubernetes objekte na osnovu YAML datoteka koje pronađe u tom direktorijumu (slično kao da ručno pozivamo `kubectl apply -f`). Uz to, Argo ima i ugrađenu veb konzolu pomoću koje možemo pratiti njegov rad, kao i aplikacije kojima on upravlja. Naše aplikacije na GitHubu će imati folder sa YAML datotekama koji opisuju Kubernetes *Deployment*, *Service* i *Ingress*, a koje će Argo pokupiti i dodati u klaster, što će pokrenuti aplikaciju i učiniti je javno dostupnom.

Kao i sve prethodne Kubernetes alate, i Argo postavljamo iz *Helm charta*. U `values` definišemo da želimo da postavimo njegove CRD-ove, kažemo mu na kom će domenu biti i postavljamo `server.insecure` parametar koji mu kaže da treba da bude dostupan samo preko HTTP-a, pošto ćemo sva HTTPS podešavanja čuvati na upravljaču ulaznog saobraćaja i tu ćemo terminirati TLS, a komunikacija između upravljača i aplikacija će biti preko HTTP-a. Odmah ispod ovoga definišemo i *Ingress* objekat (opisan `kubernetes.networking.v1.Ingress` resursom) kojim određujemo da će sav sadržaj koji stigne na `host` iz ulazne promenljive biti prosleđen na *Service* sa imenom `argocd-server` na portu koji se zove `http`, što je servis koji odgovara Argovoj veb konzoli.

Primetimo da ovde ne definišemo *cert-manager* anotaciju iz prostog razloga što još nemamo domen, tako da pokušaji da dobijemo sertifikat neće uspeti i mogu dovesti do ograničavanje usluge od strane *Let's Encrypt*-a ako nas prepozna kao maliciozni saobraćaj. Idealno, prvo bismo napravili upravljač ulaznog saobraćaja,

dodali DNS zapis za javnu IP adresu, pa se tek onda upuštali u dalji rad; kako bi to dovelo do previše „skakanja" u trenutnoj postavci, postavićemo sve bez HTTPS sertifikata trenutno, pa ćemo tu grešku ispraviti kada dodamo DNS zapise.

Nakon pravljenja ovih objekta, imamo funkcionalnu aplikaciju, ali koja i dalje ne radi ništa. Čak i ako bismo došli do nje (zamislimo da smo postavili domen), sa ovom postavkom ćemo se zaustaviti na stranici za prijavljivanje koja će nam tražiti korisničko ime i lozinku. Mogli bismo da pristupimo klasteru, nađemo *Secret* u kom Argo čuva administratorske kredencijale i prijavimo se pomoću njih, međutim jasno je da onog trenutka kada poželimo nekom drugom da damo pristup, ovo više nije zadovoljavajuće. S obzirom da ćemo naše aplikacije objavljivati na GitHubu, i to u okviru neke zajedničke organizacije, pretpostavljamo da će programeri imati naloge i biti članovi te organizacije. Idealno bi bilo kada bi oni mogli pomoću svojih GitHub naloga da se prijave na Argo da zapravo vide svoje aplikacije.

Srećom, Argo se isporučuje sa *Dex* alatom koji zna kako da se integriše sa OAuth2 snabdevačima, uključujući i GitHub. Da bismo ga podesili, potrebno je da izmenimo `argocd-cm` *ConfigMap*-u i dodamo podešavanja za *Dex*. Ova mapa je deo *chart*-a koji smo postavili i Pulumi ne zna za nju, niti njome upravlja, ali to ne znači da je ne može promeniti. Pravljenjem resursa *ConfigMapPatch* Pulumijevom Kubernetes snabdevaču kažemo da treba da izmeni postojeću (ili napravi novu) *ConfigMap*-u[11]. Na osnovu imena i imenskog prostora koji prosledimo se pronalazi odgovarajući resurs i podaci se dodaju ili spajaju. *ConfigMap* je sasvim jednostavan resurs koji umesto `spec` polja ima polje `data` i sastoji se isključivo od ključ-vrednost parova. Da bismo znali koje tačno ključeve treba da dodamo, treba da konsultujemo dokumentaciju aplikacije koja će podešavanja čitati.

U slučaju Arga, on očekuje ključ `dex.config` čija vrednost je niska u YAML formatu u kojoj se definiše tip integracije i kredencijali. Za GitHub, ovo su `clientID` i `clientSecret`, kao i opcioni niz organizacija kojima zahtevamo da korisnik pripada kako bismo mu omogućili pristup. Ove kredencijale će stek čitati iz podešavanja i ovo je jedna od retkih stvari koju je neophodno ručno napraviti, pošto GitHub ne dozvoljava automatizaciju ovog dela. Uputstvo za pravljenje OAuth2 aplikacije na GitHubu iz koje ćemo pročitati ove kredencijale se nalazi ovde: `https://docs.github.`

[11]Iako naizgled jednostavna, od prepoznavanja nedostatka do isporučivanja ove funkcionalnosti je prošlo 4 godine. Zainteresovanom čitaocu preporučujemo diskusiju na GitHubu koja u više detalja opisuje probleme tokom koncipiranja i implementacije ove funkcionalnosti i njihova rešenja: `https://github.com/pulumi/pulumi-kubernetes/issues/264`. Zanimljivo je i da Teraform trenutno nema ovoliko opšte rešenje za isti problem, već je diskusija na tu temu i dalje aktivna: `https://github.com/hashicorp/terraform-provider-kubernetes/issues/723`.

com/en/apps/oauth-apps/building-oauth-apps/creating-an-oauth-app a nešto detaljnija Argo/Dex dokumentacija ovde: https://argo-cd.readthedocs.io/en/stable/operator-manual/user-management/#dex.

Ako smo sve postavili kako treba, sada ćemo se uspešno ulogovati na Argo i videti praznu radnu površinu. Opravdano (ali pogrešno) bi bilo pomisliti da je razlog tome što zaista nismo nikakvu aplikaciju pustili u rad preko Arga. Pravi razlog je to što mi sa GitHub nalogom nemamo nikakve dozvole na Argu, pa ni ne vidimo ništa. Ovo ćemo ispraviti izmenom još jedne *ConfigMap*-e pod imenom `argocd-rbac-cm`. U njoj definišemo ključ `policy.csv`, čija vrednost definiše *rolu* koja dozvoljava *sync* akciju i proširuje *readonly* rolu koja dozvoljava pregled svih resursa, zatim je ključem `policy.default` postavljamo kao podrazumevanom za sve korisnike. Detaljniji opis kontrole pristupa u Argu, poznatom i pod akronimom *RBAC* (od *Role Based Access Control*) se nalazi u dokumentaciji: https://argo-cd.readthedocs.io/en/stable/operator-manual/rbac/.

Nažalost, najnovije verzija Arga i Deksa u trenutku pisanja ovog rada neće pravilno primeniti izmene iz *ConfigMap*-a odmah. Kako bismo ih naterali da ponovo učitaju podešavanja i sve što zavisi od njih, restartovaćemo sve podove koji pripadaju Argo *release*-u, tj. koji su u imenskom prostoru koji smo zadali. Najjednostavniji način da ovo uradimo je da ih ubijemo i oslonimo se da će ih odgovarajući *Deployment*-i i *StatefulSet* ponovo napraviti. Primetimo da je ovo vrlo imperativan zadatak, što je oštar kontrast u odnosu na deklarativan opis infrastrukture, ali Pulumi ima način da prevaziđe ovu prepreku koristeći poseban *Command* snabdevač[12]. On definiše resurse koji nam omogućavaju da izvršimo neku komandu na sistemu u trenutku pravljenja i/ili brisanja resursa, koji podržava iste opcije i čuva stanje kao i svaki drugi resurs (tako da se ista komanda neće dvaput izvršiti, čak i pri višestrukim pozivima `pulumi up`). Mi želimo da izvršimo komandu lokalno, na našem računaru, i to činimo na sledeći način:

```
import {local} from "@pulumi/command";
this.restartArgoCommand = new local.Command(`${name}_argo_restart`, {
    create: `kubectl --kubeconfig ${args.kubeconfigFile} delete pods
             --all -n ${ns}`
}, { parent: this, dependsOn: [this.argoCmPatch, this.rbacCmPatch] });
```

[12]Ovaj snabdevač je još u *preview* fazi i može doći do izmena. U ovom radu koristimo njegovu 0.10.0 verziju.

Ovde pretpostavljamo da imamo `kubectl` postavljen na računaru i njega koristimo da obrišemo sve podove (`delete pods -all`) iz imenskog prostora u kojem smo postavili Argo (`-n` argument). Jedan od argumenata ove komponente će biti i `kubeconfigFile` koji sadrži putanju do datoteke sa podešavanjima koja opisuje kako se povezujemo na Kubernetes, o čemu će biti više reči u narednom odeljku. Primetimo i da pomoću `dependsOn` opcije obezbeđujemo da će se ova komanda izvršiti nakon što izmenimo *ConfigMap*-e sa Argovim podešavanjima.

Sada se možemo pozabaviti i drugim razlogom što nam je radna površina na Argu prazna: nemamo aplikacije koje koriste Argo za isporučivanje. Ovo se može ispraviti i kroz veb konzolu, ako se ulogujemo sa administratorskim nalogom i ručno definišemo aplikacije, ali to nije dovoljno dobro rešenje za ovaj projekat. Rekli smo da u suštini Argo čita neke Git repozitorijume i na osnovu pročitanog pravi Kubernetes objekte. Ove Git repozitorijume i njihova podešavanja (koju granu posmatramo, koji poddirektorijum, u kom imenskom prostoru radimo i sl) definišemo pomoću Argovog CRD-a *Application*. Jedno rešenje bi bilo definisati *Application* za svaku našu aplikaciju u Pulumiju, međutim možemo to uraditi i fleksibilnije. Kada budemo postavljali aplikacije, napravićemo i GitHub repozitorijum čiji sadržaj će biti isključivo YAML datoteke koje definišu Argo aplikacije. U ovaj repozitorijum ćemo dodati definicije aplikacija na kojima će naši programeri raditi, ali takođe možemo dodati bilo šta što želimo da dodamo u klaster (npr. alate za monitoring, skupljače logova i sl). Poenta je da je Argo poslednja stvar koju postavljamo na klasteru kroz Pulumi, a sve buduće stvari, u bilo kom scenariju, će biti postavljene kroz Argo pomoću dodavanja odgovarajućih datoteka sa opisom Argo aplikacija na ovaj repozitorijum.

Ostaje samo da dodamo definiciju ove glavne aplikacije koju ćemo nazvati *app of apps*. U njenoj definiciji nema ničeg što je značajno drugačije od stvari koje smo do sad videli. Pravimo `CustomResource` sa `apiVersion`-om `argoproj.io/v1alpha1` i `kind`-om `Application`, smeštamo je u imenski prostor gde se nalazi i Argo (pomoću `metadata.namespace`), kao i sve resurse (druge Argo Aplikacije) koje ona definiše (pomoću `spec.destination.namespace`); te druge Aplikacije će svoje resurse smeštati u zasebne imenske prostore. Takođe kažemo Argu da želimo da automatski sinhronizuje izmene sa GitHuba u klaster, čak i ako nešto ručno izmenimo u klasteru (`spec.syncPolicy.automated.selfHeal`) ili obrišemo sa repozitorijuma (`spec.syncPolicy.automated.prune`).

Inicijalizacija snabdevača i instanciranje komponenti

Kada imamo definicije svih komponenti, treba još i da ih instanciramo. Međutim, pre nego što se upustimo u to, moramo Pulumiju nekako reći kako da se poveže na Kubernetes, kako bi mogao da napravi izmene koje smo mi opisali kodom. U prethodnom poglavlju, automatski su se povlačila podešavanja za OCI snabdevač sa sistema, ali takav pristup ovde nije moguć, s obzirom da je Kubernetes klaster tek napravljen. Uobičajeni način za opisivanje podešavanja za povezivanje na Kubernetes je *kubeconfig* datoteka, koja između ostalog sadrži adresu Kubernetes endpointa i kredencijale, ili opis kako da dođemo do njih. OCI snabdevač sadrži *get* funkciju koja će nam na osnovu identifikatora klastera (prosleđujemo ga u podešavanjima, na kraju će se ovo postavljati na osnovu izlaza *oci* steka) dati odgovarajući *kubeconfig*:

```
const kubeConfig = oci.containerengine.getClusterKubeConfigOutput({
    clusterId: config.require("okeClusterId"),
});
```

Međutim, setimo se da smo napravili privatni Kubernetes klaster i da će adresa endpointa biti neka privatna IP adresa, što znači da u ovom obliku ne možemo koristiti ova podešavanja. Ovde ćemo zahtevati od onoga ko izvršava ovaj Pulumi program da u podešavanjima prosledi IP adresu kojoj možemo pristupiti u polju `localClusterEndpoint` (npr. ako napravimo SSH tunel kroz bastion sesiju na lokalnom portu 6443, ovo će imati vrednosti `https://127.0.0.1:6443`). S obzirom da dobijena podešavanja podrazumevano traže od klijenta da proveri sertifikat koji će biti izdat za privatnu adresu, a mi se povezujemo pomoću neke druge, ovu funkcionalnost ćemo isključiti. Ovako izmenjena podešavanja ćemo snimiti u datoteku koju ćemo koristiti za izvršavanje komande za restartovanje u Argo komponenti (resurs za izvršavanje komande pripada *Command* snabdevaču, koji ne zna ništa o Kubernetesu), dok ćemo za sve ostalo instancirati Pulumi Kubernetes snabdevač:

```
import * as pulumi_kubernetes from "@pulumi/kubernetes";
const kubernetesProvider = new pulumi_kubernetes.Provider(
        "kubernetes_provider", { kubeconfig: localKubeConfig }
);
```

Konstruktor `Provider` klase liči na konstruktore resursa, s jedinom razlikom što on ne podržava dodatne opcije. Ovaj snabdevač ćemo sada uvek prosleđivati resursima, uključujući i naše komponente, na primer:

```
const ingressController = new IngressController("ingress", {
    version: config.require("ingressVersion"),
    cpuRequest: config.require("ingressCpuRequest"),
    memoryRequest: config.require("ingressMemoryRequest"),
    maxReplicas: config.requireNumber("ingressMaxReplicas"),
}, {
    provider: kubernetesProvider
});
```

Pošto svi resursi koje komponente definišu u opcijama prosleđuju `parent: this`, oni će „naslediti" snabdevača kojeg prosleđujemo komponentama i podrazumevano njega koristiti za Kubernetes resurse. Kada instanciramo *cert-manager* i Argo, dodajemo `dependsOn` opciju postavljenu na `ingressController`, kako bi sačekali da se upravljač ulaznog saobraćaja napravi, da bi se oni i njihovi *Ingress* objekti pravilno instancirali.

Cloudflare

Treći snabdevač koji ćemo koristiti u ovom projektu je *Cloudflare*, koji iako je primarno poznat kao mreža za dostavljanje sadržaja (*Content Delivery Network*), nama će služiti samo kao DNS. Pretpostavljamo da imamo nalog na ovoj platformi i već povezan i postavljen domen. Kako bi se Pulumi povezao sa *Cloudflare*-om, potrebno je generisati *API Token* i dodati ga u podešavanja pod ključem `cloudflare:apiToken`, koji će snabdevač automatski čitati. Uputstvo za dobijanje tokena je dostupno ovde: `https://developers.cloudflare.com/fundamentals/api/get-started/create-token/`. Druga podešavanja ćemo komentarisati kad dođemo do njih u kodu.

Ovo je ubedljivo najjednostavniji stek u projektu i ne definiše nikakve komponente, već se sve nalazi u `index.ts` datoteci. Prvo, na osnovu domena koji nam je prosleđen u podešavanjima dohvatamo *zonu* koja je analogna *compartment*-u na OCI-ju i uvek je vezana za domen (može postojati više *Cloudflare zona* za isti domen, ali samo jedna može biti aktivna):

```
const domain = config.require("domain");
const zones = cloudflare.getZonesOutput({
    filter: {
        name: domain,
```

```
        status: "active",
    },
});
```

Zatim, pravimo DNS zapis u toj zoni koji će biti usmeren na javnu IP adresu upravljača kojeg smo napravili u prethodnom odeljku:

```
const dnsRecordName = config.get("subdomain") ?
        `*.${config.get("subdomain")}` : "*";
const dnsRecord = new cloudflare.Record("dns_record", {
    zoneId: zones.apply(zs => zs.zones[0].id!),
    name: dnsRecordName,
    value: config.require("ip"),
    type: "A",
    proxied: false,
});
```

Ako je u podešavanjima postavljen ključ `subdomain`, pravimo zapis koji sve poddomene drugog nivoa povezuje sa adresom iz parametra `ip`, pa će adresa Arga biti oblika *argo.master.projekat.com*. Ako ne prosledimo ovaj parametar, sve poddomene povezujemo sa prosleđenom IP adresom i pomenuta adresa bi u tom slučaju bila *argo.projekat.com*. U oba slučaja domen koji određuje zonu je *projekat.com*. Argumentom `proxied` kažemo Cloudflare-u da ne rutira zahteve kroz svoju mrežu, pošto ćemo naići na probleme sa SSL-om ako koristimo poddomene drugog nivoa [2]. Nakon ovoga dodajemo i *Page Rule* instanciranjem `cloudflare.PageRule` resursa kojim isključujemo SSL podršku za sve zahteve koji idu na ovaj poddomen. Ovo neće zapravo onemogućiti HTTPS, već će samo reći *Cloudflare*-u da ne pokušava da šifruje ništa nego ih samo propusti ka našim serverima, koji će ispravno da rukuju HTTPS saobraćajem.

Problem je zapravo to što *wildcard* sertifikati (koji pokrivaju sve poddomene) za poddomene drugog nivoa nisu podržani nigde, a *Cloudflare* kao CDN mora da terminira TLS kada dobije HTTP zahtev, što nije moguće ako ne može da dobije sertifikat čiji host odgovara imenu iz DNS zapisa. Nama ovo ne predstavlja problem na upravljaču ulaznog saobraćaja, jer mi nikad zapravo nećemo praviti *wildcard* sertifikate, već će *cert-manager* dinamički tražiti sertifikate samo za one poddomene koje koristimo. U teoriji bismo mogli da dinamički pravimo DNS zapise za svaku *Ingress* instancu, ali bi ovo bilo **daleko** kompleksnije.

Ostaje nam još jedna stvar koju nismo uradili u sklopu Kubernetes steka, a to je anotiranje Argo *Ingress*-a kako bi mogao da dobije TLS sertifikat. Ovo logički ne pripada ovom steku, već se ovde nalazi isključivo iz praktičnih razloga. Dopustićemo da se ovaj parametar ne prosledi i u tom slučaju ne radimo ništa. Međutim, ako se prosledi, očekujemo da u podešavanjima imamo i (lokalni) *kubeconfig*, naziv *Ingress* objekta kog treba da anotiramo i naziv *ClusterIssuer*-a. Koristeći sve ove podatke, pravimo Kubernetes snabdevač na sličan način kao u Kubernetes steku, samo bez obrade prosleđenog *kubeconfig*-a i sa parametrom koji dozvoljava snabdevaču da pri brisanju smatra resurse obrisanim ako nije moguće doći do Kubernetesa (npr. ako prvo obrišemo Kubernetes, pa onda ovaj stek). Pomoću ovoj snabdevača pravimo resurs tipa *IngressPatch* koji se ponaša potpuno analogno *ConfigMapPatch* resursu, ali za *Ingress*.

Ako je sve u redu, nakon što izvršimo `pulumi up` ćemo moći u pretraživaču da otvorimo stranicu `https://argo.PODDOMEN.DOMEN` gde će nas dočekati stranica za prijavljivanje i da se uspešno prijavimo pomoću GitHub naloga. Nakon toga, ako smo stekove podizali onim redom kojim su izloženi u ovom radu, na radnoj površini bi trebalo da vidimo *app-of-apps* aplikaciju koja prijavljuje grešku jer ne može da nađe repozitorijum iz svojih podešavanja.

GitHub

U ovom odeljku ćemo napraviti sve potrebne repozitorijume i pružiti Argu „materijal za rad". Još kad smo postavljali autentifikaciju na Argu smo pretpostavili da postoji neka organizacija na GitHubu čiji članovi su svi programeri koji će razvijati aplikacije koje treba da se pokrenu na klasteru. Sada ćemo dodati još jednu pretpostavku: u toj organizaciji se nalazi projekat koji predstavlja „šablon" ili „skelet" za sve ostale aplikacije i on sadrži definiciju Kubernetes *Ingress* objekta na putanji `k8s-resources/ingress.yaml`. Sve druge repozitorijume ćemo praviti kao kopije ovog. Sama struktura repozitorijuma može biti proizvoljna, kao i sam projekat koji može biti pisan u bilo kom programskom jeziku i koristiti bilo koji razvojni okvir.

Organizacija koju ćemo koristiti za primer u ovom radu je `lukaj-master` i u njoj se može naći *Hello World* aplikacija u *ASP.NET* razvojnom okviru: `https://github.com/lukaj-master/aspnet-template-app`. U direktorijumu `TemplateApp` se nalazi kod aplikacije i *Dockerfile* za izgradnju slike; ovde je bitno setiti se da ćemo aplikacije isporučivati na mašine (Kubernetes čvorove) koji koriste ARM procesor i napraviti adekvatnu sliku (proces izgrađivanja slike, s druge strane, pretposta-

vljamo da će biti na AMD64 infrastrukturi). U `k8s-resources` se nalaze definicije Kubernetes *Deployment*-a, *Service*-a i *Ingress*-a, a u `.github/workflows` definicija GitHub akcije koja će na svaki *push* na master granu da izgradi sliku, postavi je na GitHub-ov repozitorijum za slike (GHCR) i izmeni *k8s-resources/deployment.yaml* tako da referencira novu sliku. Ovo omogućava da programer samo postavi svoj kod na GitHub, a ostatak procesa izgradnje i isporučivanja je potpuno automatski i može se pratiti kroz GitHub i Argo.

Vratimo se sada Pulumi projektu i to konkretno jedinoj komponenti koju GitHub stek definiše, `TeamWithRepo`. Ona enkapsulira nekoliko resursa: repozitorijum koji će biti napravljen iz (nekog) šablonskog repozitorijuma, tim koji će biti napravljen u organizaciji, skup članova tima (predstavljenih njihovim korisničkim imenima) koji moraju biti članovi organizacije, i daje dozvolu novom timu da može da objavljuje kod na novom repozitorijumu upotrebom `TeamRepository` resursa. Ovaj kod ne koristi ništa što nismo u prethodnim odeljcima već videli.

Sadržaj korenog direktorijuma steka je nešto zanimljiviji, s obzirom da možemo primetiti neke nove datoteke u poređenju s drugim stekovima. Prvo, u `argo-app-template.yaml` se nalazi definicija Kubernetes objekta koji opisuje Argo aplikaciju. Ova datoteka će se menjati za svaku aplikaciju tako da sadrži konkretno ime i URL repozitorijuma i kopirati u repozitorijum koji će sadržati sve definicije Argo aplikacija. Druga „nova" datoteka je `utils.ts`, u koju smo smestili pomoćne funkcije koje rade s Git repozitorijumima, i to:

- `editIngressDefs(repoPath: string, templateHost: string, appHost: string)` klonira repozitorijum (aplikacije), menja datoteku koja opisuje *Ingress* resurs tako da postavlja poddomen na odgovarajući poddomen aplikacije, pravi Git *commit* i objavljuje izmene.

- `populateArgoAppRepo(repoPath: string, appNames: string[], appCloneUrls: string[])` uzima putanju do praznog repozitorijuma i nazive i URLove aplikacija, popunjava ga sa definicijama Argo aplikacija, pravi Git *commit* i objavljuje izmene.

Obe funkcije koriste `simple-git` biblioteku za rad sa Gitom, koja pretpostavlja da na računaru postoji postavljen `git`.

Druga veća promena u odnosu na ostale stekove je upotreba datoteke za praćenje struktuiranih podataka, konkretno spiska timova, njihovih članova i repozitorijuma. I dalje koristimo Pulumi podešavanja za jednostavne tipove (uključujući i

putanju do ove datoteke), ali s obzirom na nešto kompleksniju strukturu koju čovek mora da menja i održava, ovde je zgodnije koristiti datoteku koja služi samo toj svrsi. Primer ove datoteke se može naći u korenom direktorijumu repozitorijuma pod nazivom `org-chart.yaml`. Nakon što pročitamo i isparsiramo sadržaj ove datoteke, pravimo instancu snabdevača.

Pulumi GitHub snabdevač zahteva token naloga koji će koristiti za pristup API-ju koji je potrebno izgenerisati (uputstvo za ovaj postupak je dostupno na `https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens`) i u našem slučaju naziv organizacije na koju će se odnositi sve operacije. Slično kao Kubernetes snabdevač, i ovaj ćemo prosleđivati svim resursima.

Prvo skupljamo sva korisnička imena iz svih timova i za svaki pravimo instancu resursa `Membership` koja će pozvati korisnika s tim korisničkim imenom u organizaciju. Zatim, pravimo prazan repozitorijum koji će sadržati definicije svih Argo aplikacija, `argo-app-of-apps`. Nakon toga, za svaki tim definisan u datoteci pravimo instancu *TeamWithRepo* komponente i prosleđujemo joj tražene podatke.

Možemo da primetimo da ova komponenta „zavisi" (definiše `dependsOn` opciju) od prethodno napravljenog repozitorijuma za Argo aplikacije bez nekog očiglednog razloga. Ovo je deo zaobilaznog rešenja za problem uzrokovan jednom specifičnošću Pulumijevog GitHub snabdevača: sve izlazne promenljive repozitorijuma se popunjavaju odmah, a ne nakon što se resurs zapravo napravi. Ovo znači da kada želimo da izvršimo `populateArgoAppRepo` kako bi dodali datoteke u repozitorijum, mi zapravo nemamo način da obezbedimo da će se ovo desiti nakon što se repozitorijum napravi, pošto na običan (imperativni) kod ne možemo staviti `dependsOn` ograničenje (Pulumi koristi imperativne programske jezike, ali je i dalje u suštini deklarativan sistem). Umesto toga, stavljamo `dependsOn` na `TeamWithRepo` resurs, čiji `Team` resurs sadrži polje `etag` koje se popunjava tek kada se tim zapravo napravi. Konstrukcijom

```
pulumi.all(teams.map(t=>t.team.etag)).apply(() => {
    ...
})
```

obezbeđujemo da se kod unutar vitičastih zagrada izvrši samo kada su svi timovi napravljeni, što tranzitivno znači da je i `argo-app-of-apps` repozitorijum napravljen. Primetimo i da su funkcije koje koristimo ovde idempotentne, tako da višestru-

ko izvršavanje (recimo, ako menjamo stek pa više puta pokrećemo `pulumi up`) ne predstavlja problem.

Pokretanjem `pulumi up` na ovom steku će napraviti repozitorijume po opisu iz `org-chart.yaml` datoteke i poslati pozive drugim GitHub nalozima po potrebi. Pokretanjem `pulumi down` (ekvivalentno, `pulumi destroy`) će svi repozitorijumi biti bespovratno obrisani, a nalozi izbačeni iz organizacije; ovo verovatno nikada ne želite da radite van faze testiranja.

3.5 Pulumi program kao deo konvencionalnog programa

Poslednji deo ovog projekta je spajanje stekova opisanih u prethodnom poglavlju u jedan *Node.js* program. Za ovu potrebu, koristićemo deo Pulumija koji se naziva *Automation API* i koji definiše različite klase i metode u `@pulumi/pulumi/automation` paketu. Ovaj program će biti namerno minimalistički i služiti kao demonstracija koncepta, s obzirom da su mogućnosti praktično neograničene, a potrebe će izuzetno varirati od organizacije do organizacije. Stoga, napravićemo konzolnu aplikaciju koristeći *Node.js*.

U korenom direktorijumu repozitorijuma, osim standardne postavke *Node.js* projekta koji koristi TajpSkript i već opisane `org-chart.yaml` datoteke koju prosleđujemo GitHub steku, imamo `pulumi.ts` i `utils.ts` datoteke sa izvornim kodom. U `utils.ts` imamo nekoliko pomoćnih funkcija: omotač za ispis na standardni izlaz koju ćemo moći da prosleđujemo kao *callback* (metode se ne mogu prosleđivati, pošto `this` menja vrednost u zavisnosti od konteksta), funkciju za uspostavljanje SSH tunela koja izvršava `ssh` komandu sa odgovarajućim parametrima u zasebnom procesu na mašini i funkciju koja ubija proces, koja će se koristiti za ubijanje tunela.

Sva logika programa se nalazi u `pulumi.ts` datoteci. Pomoću konstanti smo definisali da radimo na domenu `luka-j.rocks` i njegovom poddomenu `master` (domen je potrebno prethodno registrovati i podesiti *Cloudflare* za njega; poddomen može biti proizvoljan), kao i lokacije stekova i `org-chart` datoteke. Program će zahtevati dva argumenta komandne linije, gde će prvi predstavljati komandu koju želimo da izvršimo, a drugi naziv steka na kom radimo, bez prefiksa koji određuje tip steka (npr. `dev` će se odnositi na stekove `gh_dev`, `oci_dev`, `k8s_dev` i `cf_dev`, čija podešavanja trenutno imamo u projektu). Rekli smo pri početku ove glave da je za čuvanje i čitanje poverljivih podešavanja potrebna lozinka, a da je ne bismo stalno unosili,

sačuvaćemo je u datoteci `pulumi_passphrase` koju ne dodajemo na repozitorijum i postavićemo promenljivu okruženja `PULUMI_CONFIG_PASSPHRASE_FILE` na putanju do ove datoteke, koju će Pulumi implicitno čitati.

U ovom programu podržavamo dve glavne komande: `up` i `down` ili `destroy`, s tim da druga ima i varijantu `destroyAll` koja će obrisati sve resurse, uključujući i GitHub repozitorijume, za razliku od `down` i `destroy` koji će ovaj korak preskočiti. Komanda `up` podiže svu infrastrukturu tako što inicijalizuje četiri steka, i to ovim redom:

1. Podižemo GitHub stek (funkcija `upGithubStack`), njemu se prosleđuje putanja do `org-chart` datoteke, a iz njegovih izlaza izdvajamo naziv organizacije i putanju do `argo-app-of-apps` repozitorijuma.

2. Podižemo OCI stek (funkcija `upOciStack`) koji ne zahteva nikakva dodatna podešavanja, a iz njegovih izlaza uzimamo identifikator Kubernetes klastera, Kubernetes endpoint i identifikator bastion sesije.

3. Na osnovu identifikatora bastion sesije i adrese endpointa pravimo SSH tunel pomoću `startSshTunnel` funkcije iz `utils`-a. Ovo moramo uraditi pre nego što krenemo da radimo bilo šta sa Kubernetesom.

4. Podižemo Kubernetes stek (funkcija `upK8sStack`) koji od prethodnih stekova zahteva identifikator Kubernetes klastera, naziv GitHub organizacije i putanju do `argo-app-of-apps` direktorijuma (kako bi podesio Argo), a od njegovih izlaza nas zanimaju javna IP adresa upravljača ulaznog saobraćaja, naziv *ClusterIssuer* objekta, lokalni *kubeconfig* koji je napravljen unutar ovog steka i naziv *Ingress* objekta koji odgovara Argovoj veb konzoli.

5. Na kraju, podižemo *Cloudflare* stek, kome prosleđujemo sve što smo izdvojili kao relevantne izlaze Kubernetes steka.

Na kraju `up` metode stopiramo SSH tunnel (ako je napravljen), a pošto ovo treba da se izvrši čak i ako dođe do greške, nalazi se u `finally` bloku. Da bismo mogli bilo šta da radimo sa stekom, prvo moramo dobiti referencu na njega, a s obzirom da je u pitanju stek koji imamo u datotekama na našoj mašini koristimo `LocalWorkspace`:

```
await LocalWorkspace.createOrSelectStack({
    workDir: GITHUB_STACK_DIR,
```

```
    stackName: `gh_${stackName}`
});
```

Ovaj objekat je tipa `Stack` i ima metode koje odgovaraju komandama koje bismo izvršavali pomoću `pulumi` CLI alata. Ulazne promenljive za stek će uvek biti podešavanja (*configuration*), koja se može postaviti pomoću `setConfig()` metode na dobijenom `Stack` objektu tako što prosledimo naziv parametra podešavanja i, ako vrednost nije poverljiva, objekat koji se sastoji od ključa `value` i vrednosti parametra (ako vrednost jeste poverljiva, koristi se ključ `secret` umesto `value`). Ove vrednosti dolaze ili od nekog drugog steka ili su u pitanju literali ili konstante iz programa. Pozivom `setConfig()` se menja datoteka koji predstavlja podešavanja za određeni stek i prepisuju postojeće vrednosti za ključ (ako postoje). Nakon što postavimo sva podešavanja, pozivamo metodu `up` sa opcijom koja kaže da pri svakom redu izlaza treba pozvati funkciju `plainLog` (omotač oko ispisa na standardni izlaz) koja omogućava ispit na standardni izlaz u realnom vremenu.

Metoda `up()` će nam vratiti objekat tipa `UpResult` koji u `outputs` polju sadrži sve izlaze steka. Izlazi će najčešće biti tipa `Output`, međutim pošto znamo da su u trenutku kada se `up()` završi svi resursi već napravljeni, bezbedno je direktno pristupiti njegovom `value` polju i pročitati podatke koje nas zanimaju. Sličnu logiku koristimo za sve stekove, jedino što se razlikuje je skup parametara podešavanja koje postavljamo i izlaznih promenljiva koje izdvajamo.

Funkcija `down()` koja uništava infrastrukturu je značajno jednostavnija. Na isti način dohvatamo `Stack` objekte, samo ovaj put nad njima pozivamo `destroy()` metodu bez dodatnih opcija. U zavisnosti od toga da li je komanda `destroyAll` ili ne, prosleđuje se logički parametar koji omogućava preskakanje GitHub steka u ovom procesu. Primetimo da nema potrebe zasebno uništavati Kubernetes stek, s obzirom da će se uništavanjem OCI steka svakako sve vezano za Kubernetes obrisati.

Funkcija koja bi izvršavala posao analogan `pulumi preview` CLI komandi (koja je ekvivalent `terraform plan`-u) nije implementirana u sklopu ovog projekta. `Stack` sadrži `preview()` metodu i to će raditi nesmetano za sve stekove sem za Kubernetes. U ovom slučaju je neophodno napraviti SSH tunel kako bismo mogli da instanciramo snabdevač koji je neophodan `preview()` metodi, što zahteva da imamo nekakav bastion postavljen, pretpostavka koja ne mora biti uvek ispunjena (npr. na početku, ali i u raznim situacijama gde može doći do greške u sred rada). S obzirom na komplikacije koje bi implikacija dodala i činjenicu da ona nije usko vezana za poentu ovog rada, ovo ostavljamo za budući rad.

Glava 4

Zaključak

Iskoristimo poslednjih par stranica da se osvrnemo na širu sliku i šta smo sve uradili, posebno u prethodnoj glavi s obzirom da ona sadrži značajnu količinu originalnog rada. Prve dve glave su bile svojevrsna ekspozicija i teorijski uvod i kao takve nadamo se da su bile korisne čitaocu.

4.1 Pregled i značaj urađenog

Maltene sve tehnologije koje smo u ovom radu koristili su relativno nove i način na koji se one koriste u realnom svetu i dalje brzo evoluira. Mi smo ovde predstavili jedan predlog na koji ovi alati, a konkretno Pulumi i njegov *Automation API*, mogu da se koriste da se opišu i naprave konceptualno netrivijalne infrastrukture. Smatramo da su ovi alati u potpunosti adekvatni i za podešavanje Kubernetesa ili sličnih servisa na infrastrukturi u oblaku i da sve prednosti koji oni imaju za tradicionalnu infrastrukturu važe i za Kubernetes, te da oni značajno olakšavaju posao administratorima sistema. Takođe verujemo da su dobra alternativa za podešavanje pratećih, spoljnih servisa, poput prikazanih GitHuba i *Cloudflare*-a, posebno u slučajevima kada nije neophodna duboka ili izuzetno specifična integracija koja bi zahtevala komplikovanu imperativnu logiku i opravdala dodatan trošak vremena za upoznavanje sa API-jem i održavanjem još jedne heterogene komponente u sistemu.

Predstavljena organizacija programa u nekoliko odvojenih stekova, koji sve resurse koji predstavljaju logičku celinu ili imaju komplikovaniji skup argumenata izdvajaju u komponente, predstavlja idealan stepen modularizacije po našem mišljenju. Sasvim je realno pretpostaviti da će budući razvijači ili korisnici ovog programa hteti da infrastrukturu u oblaku postavi pomoću drugog snabdevača, ili da

ne koristi GitHub ili *Cloudflare*, i moramo omogućiti da se ove izmene mogu napraviti na što jednostavniji način. Pretpostavljamo da će nešto ređe, ili u manjoj meri, postojati potreba za menjanjem konkretnih delova infrastrukture, ali s obzirom da ni to ne smatramo retkom operacijom, i u tim slučajevima želimo da pružimo što veću udobnost pomoću komponenti koje uvode što manje pretpostavki o spoljnim okolnostima.

Što se samog sadržaja projekta tiče, najvećim uspehom smatramo to što naš program podiže celu infrastrukturu bez ikakve interakcije s korisnikom (pod pretpostavkom da su prateći servisi pravilno inicijalizovani). Važno je da postoji mogućnost da se ponašanje promeni kroz brojne podesive parametre, ali i da podrazumevani, koji su deo projekta, omogućavaju podizanje funkcionalnog sistema. Ovakav sistem može se koristiti od strane bilo koje organizacije koja broji nekoliko (ili nekoliko desetina) timova, ali i npr. kao infrastruktura za studentske projekte koji su obliku veb aplikacija. On omogućava standardizovano okruženje za isporučivanje uz minimalan napor od strane programera, iz čije perspektive je logika izgradnje i puštanja u rad aplikacije potpuno automatizovana.

4.2 Budući rad

Postoji nekoliko pravaca u kome se ovaj projekat može proširiti ili poboljšati. U „širinu", dodavanjem novih komponenti ili stekova za česte funkcionalnosti. Mi se nismo bavili postavljanjem baze podataka, sistema za keširanje, redova događaja i sl, što je sve očekivano u ozbiljnijim sistemima. Takođe, može se dodati podrška za druge, popularnije, snabdevače infrastrukture u oblaku, kao i dodatne i/ili alternativne prateće servise, što će ovaj projekat učiniti korisnijim većem skupu ljudi. Mi takođe nismo posvetili mnogo pažnje Kubernetes alatima, poput onih za monitoring (npr. *Prometheus, Alertmanager, Grafana*), skupljanje logova (npr. *FluentBit*), komunikaciju između servisa (npr. neki *service mesh*, poput *Istio* ili *Linkerd*), naprednu logiku za skaliranje (npr. *KEDA*) ili polise (npr. *Kyverno*; zainteresovani čitalac može da dobije dodatne ideje istraživanjem ove stranice: `https://landscape.cncf.io/`), ali jesmo postavili osnovu u obliku `app-of-apps` repozitorijuma koji se može koristiti za dodavanje svega navedenog.

Veće organizacije sa više ljudi koji treba da imaju mogućnost upravljanja infrastrukturom će takođe želeti da postave korisničke naloge i adekvatne IAM dozvole za njih na snabdevaču infrastrukture u oblaku, što može biti odvojena komponenta

u OCI steku. U tom slučaju takođe želimo da koristimo neki deljeni *storage backend* za stanje, umesto podrazumevanih lokalnih datoteka.

Drugi pravac je unapređivanje onog što već postoji. Format `org-chart` datoteke trenutno odgovara ograničenom broju slučaja upotreba i može se učiniti fleksibilnijim, ili upotrebiti neki napredniji način skladištenja ili povlačenja podataka. Nešto ambiciozniji poduhvat bi bio unapređivanje aplikacije koja se koristi za upravljanje stekovima. Primitivna konzolna aplikacija radi posao kao demonstracija koncepta, ali kako bi ceo projekat bio prijemčiviji, valjalo bi napraviti i jedinstven grafički interfejs pomoću koga se može pregledati trenutno stanje, upravljati podešavanjima i izvršavati akcije za postavku i uništavanje infrastrukture. Ovo bi bio programerski projekat koji je izazov za sebe, ali za koji smo u ovom radu dali sve neophodne delove koji se tiču same logike za upravljanje infrastrukturom.

Bibliografija

[1] AWS. What is AWS?, 2024. on-line at: https://aws.amazon.com/what-is-aws/.

[2] Cloudflare. ERR_SSL_VERSION_OR_CIPHER_MISMATCH: Multi-level subdomains, 2024. on-line at: https://developers.cloudflare.com/ssl/troubleshooting/version-cipher-mismatch/#multi-level-subdomains.

[3] Terraform developers. Terraform 1.6 Release Notes, 2023. on-line at: https://github.com/hashicorp/terraform/blob/v1.6/CHANGELOG.md.

[4] Joe Duffy. Pulumi 1.0, 2019. on-line at: https://www.pulumi.com/blog/pulumi-1-0/.

[5] Sean Gillespie. Pulumi GitHub: registerOutputs usage is not clear, 2019. on-line at: https://github.com/pulumi/pulumi/issues/2653#issuecomment-484956028.

[6] HashiCorp. CDK for Terraform, 2024. on-line at: https://developer.hashicorp.com/terraform/cdktf.

[7] Mitchell Hashimoto. The Story of HashiCorp Terraform with Mitchell Hashimoto, 2021. on-line at: https://www.hashicorp.com/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto.

[8] Vincent Hoogendoorn and David Fowler. Aspire to become a full fledged IaC product, or focus on integration with IaC products?, 2024. on-line at: https://github.com/dotnet/aspire/discussions/2014.

[9] Kubernetes. Kubernetes Overview, 2023. on-line at: https://kubernetes.io/docs/concepts/overview/.

[10] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[11] Cyrus Najmabadi. Pulumi GitHub: registerOutputs usage is not clear, 2019. on-line at: `https://github.com/pulumi/pulumi/issues/2653#issuecomment-484959592`.

[12] Gergely Orosz. Handling a Regional Outage: Comparing the Response From AWS, Azure and GCP, 2023. on-line at: `https://newsletter.pragmaticengineer.com/p/handling-a-regional-outage-comparing`.

[13] Pulumi. Pulumi Terraform Bridge, 2024. on-line at: `https://github.com/pulumi/pulumi-terraform-bridge`.

[14] Yahoo! Press Release. Yahoo! to acquire GeoCities, 1999. on-line at: `https://web.archive.org/web/20060502040740/http://yhoo.client.shareholder.com/press/ReleaseDetail.cfm?ReleaseID=173652`.

[15] Felix Richter. Amazon maintains cloud lead as microsoft edges closer. 2024. on-line at: `https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/`.

[16] David Sandilands. *Puppet 8 for DevOps Engineers*, chapter Puppet's history and relationship to DevOps. Packt, 2023.

[17] Daniel Sokolowski and Guido Salvaneschi. Towards reliable infrastructure as code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 318–321. IEEE, 2023.

[18] Terraform. Terraform Providers, 2024. on-line at: `https://registry.terraform.io/browse/providers`.

Biografija autora

Luka Jovičić je rođen 16. aprila 2000. godine u Pančevu. Još u osnovnoj školi se zainteresovao za programiranje, kada je naučio programski jezik BASIC, a zatim C, i bio nagrađivan na državnim takmičenjima iz informatike. 2014. godine upisuje Matematičku gimnaziju u Beogradu, a narednih godina u tri navrata učestvuje na konkursu za Android aplikacije Telekoma Srbije na kojima njegove aplikacije bivaju nagrađivane, a 2018. godine osvaja Specijalnu nagradu za inovativnost „Igor Osmokrović" na Regionalnom app izazovu u organizaciji iste kompanije.

2018. godine upisuje smer Informatika na Matematičkom fakultetu Univerziteta u Beogradu. Od avgusta 2019. radi kao softverski inžinjer u kompaniji *Centili*, gde se primarno bavi razvojem *backend* aplikacija, ali ima priliku da radi i na okruženjima u oblaku i njihovoj postavci. 2022. godine sa prosečnom ocenom 9,13 stiče zvanje diplomiranog informatičara na Matematičkom fakultetu. U akademskoj 2022/23. godini radio je kao saradnik u nastavi na istom fakultetu, gde je držao vežbe na predmetima Računarske mreže i Programiranje baza podataka. Od jula 2024. radi u kompaniji *Pure Storage* u Pragu, gde razvija softver za fleš diskove koji se koriste u data centrima.