Univerzitet u Beogradu
Matematički fakultet

Master rad

Razvoj REST servisa u stilu arhitekture "bez servera" na platformi Microsoft Azure

Miloš Milovanović

Mentor: **prof. dr Vladimir Filipović**
Matematički fakultet, Univerzitet u Beogradu

Članovi komisije: **prof. dr Saša Malkov**
Matematički fakultet, Univerzitet u Beogradu

dr Aleksandar Kartelj
Matematički fakultet, Univerzitet u Beogradu

***Datum odbrane*: _____**

Sažetak

Ovaj rad se bavi predstavljanjem koncepata arhitekture “bez servera” i modela “funkcija kao servis” na platformama za računarstvo u oblaku. U radu je prikazan pregled platforme *Azure Functions* koja se zasniva na ovom modelu. Takođe prikazana je implementacija REST servisa “Recepti API” koji omogućava upravljanje kulinarskim receptima, namirnicama i njihovim nutritivnim informacijama. Smisao implementacije servisa je da na praktičan način prikaže koncepte i principe arhitekture “bez servera” i platforme *Azure Functions* koje su prikazane u radu.

Ključne reči: računarstvo u oblaku, arhitektura “bez servera”, “funkcija kao servis”, *Microsoft Azure*, servisi REST

Sadržaj

1 Uvod

Isporučioci javnih platformi za računarstvo u oblaku od njihovog nastanka beleže stalni rast u korišćenju svojih usluga. Poslednjih godina primetno je da je ovaj rast značajno uvećan. Celokupno tržište servisa računarstva u oblaku poraslo je za 17.5% u toku 2019. godine u odnosu na prethodnu po istraživanju kompanije *Gartner* [1]. Sa porastom popularnosti i dostupnosti ovih servisa, mnoge kompanije se odlučuju da hostovanje svog softvera delom ili u potpunosti premeste sa interne infrastrukture na neke od platformi u oblaku. Programeri sve češće prate pristup kod kog se razvoj novih aplikacija primarno bazira na platforama u oblaku (*eng. Cloud native*). Isporučioci platformi sa druge strane teže da odgovore na zahteve tržišta većom i raznovrsnijom ponudom servisa.

Računarstvo “bez servera" (*eng. Serverless computing*) je jedan od novijih modela servisa računarstva u oblaku kod kog je isporučilac primarno zadužen za upravljanje delom arhitekture na serverskoj strani. Razvojem koncepata kao što su kontejneri (*eng. Containers*) i njihovo organizacija (*eng. Container Orchestration*) stvorili su se uslovi da se upravljanje infrastrukturnim resursima može obavljati na apstraktniji i automatizovan način. Sa tim dolazi i do stvaranja potpuno novog načina izvršavanja na platformama u oblaku pod nazivom “funkcija kao servis” (*eng. Function as a Service*). Računarstvo “bez servera" i “funkcija kao servis” su poslednjih godina vrlo popularni termini i kod velikih isporučilaca platformi u oblaku, a takođe i u zajednici otvorenog koda. Vodeće kompanije u ovoj oblasti kao što su *Amazon*, *Microsoft* i *Google* imaju u svojoj ponudi sada već relativno zrele platforme na ovom modelu, a takođe primetan je napredak i kod drugih isporučilaca.

Cilj ovog rada je da prikaže računarstvo “bez servera" i model “funkcija kao servis” kao jedan od pristupa u softverskoj arhitekturi na platformama u oblaku. Pored toga, da prikaže način rada, pregled funkcionalnosti i praktičnu primenu *Azure Functions* kao jedne od tri najpopularnije platforme ovog tipa.

U drugom poglavlju rada opisan je nastanak i razvoj računarstva “bez servera", od slojevite i mikroservisne arhitekture, kroz pojavu virtualizacije zasnovane na kontejnerima do računarstva u oblaku i modela servisa. Objašnjeni su teorijski aspekti arhitekture “bez servera" i modela izvršavanja “funkcija kao servis”, njihove glavne osobine, prednosti i nedostaci.

U trećem poglavlju dat je kratak prikaz platformi “funkcija kao servis”. Pored toga, detaljno je opisana platforma *Azure Functions* od načina razvoja funkcija, projekata funkcija, okidača i vezivanja, do internog načina izvršavanja i postavljanja na platformu.

U četvrtom poglavlju će biti demonstrirana implementacija servisa REST API korišćenjem arhitekture i tehnika opisanih u prethodna dva poglavlja. Servis koji će biti prikazan zamišljen je kao deo servisa “Recepti API" za kulinarske recepte. Biće dati funkcionalni opis, model baze podataka i značajni delovi koda, a celokupan kod servisa biće dostupan javno na adresi https://github.com/milosmi11166/Master. Biće prikazano rezervisanje resursa, postavljanje i testiranje servisa na platformi u oblaku *Microsoft Azure*.

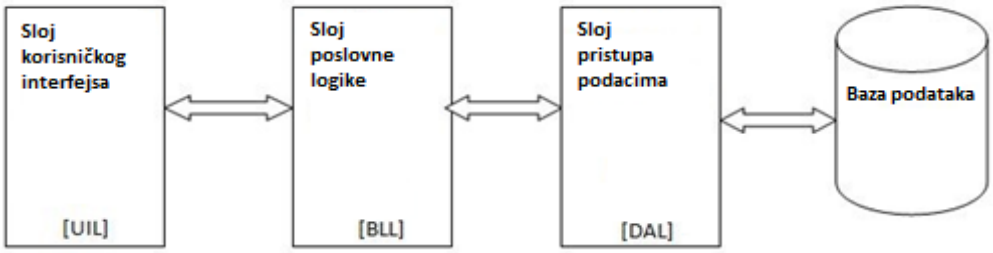
U poslednjem petom poglavlju biće izveden zaključak.

2 Pregled nekih arhitektura i računarstvo “bez servera”

U ovom poglavlju predstavljen je pregled nekih od aktuelnih arhitektura i tehnologija za razvoj veb servisa i aplikacija. Pored toga, opisani su glavni modeli servisa računarstva u oblaku i dati koncepti računarstva “bez servera”. Detaljnije je predstavljen model “funkcija kao servis”, kao jedan od oblika računarstva “bez servera”, dat je njegov način funkcionisanja, osobine, prednosti i nedostaci.

2.1 Monolitna, slojevita i mikroservisna arhitektura

U domenu veb aplikacija, monolitnim aplikacijama nazivamo jednoslojne aplikacije koje u sebi sadrže čvrsto vezane komponente, kao što su korisnički interfejs i komunikacija sa sistemima za čuvanje podataka. Ovakve aplikacije se sastoje se iz jedne ili nekoliko usko povezanih izvršnih datoteka koje se najčešće postavljaju na virtualnim mašinama ili direktno na operativnom sistemu serverskog računara. U slučaju kada su komponente jasno odvojene i labavo vezane, koristi se i naziv slojevita arhitektura (*eng. N-Tier architecture*). Čest primer je troslojna arhitektura, koja aplikacije deli na sloj korisničkog interfejsa, aplikativni sloj i sloj podataka. Sloj korisničkog interfejsa je zadužen za interakciju sa korisnikom, aplikativni sloj za implementaciju poslovne logike, a sloj podataka za implementaciju pristupa sistemima za skladištenje podataka, kao što je prikazano na Slici 1.



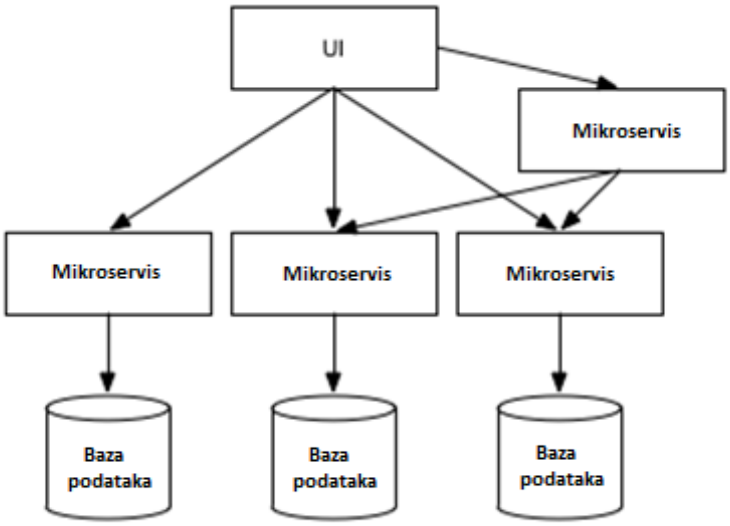
Slika 1. Komponente troslojne arhitekture

U slučaju srednje složenih aplikacija slojevita arhitektura ima dosta prednosti zbog svoje jednostavnosti za razvoj, testiranje i postavljanje u produkciju. Kod većih i

kompleksnijih aplikacija pojavljuju se neki od nedostataka ovog pristupa. Sa većim brojem funkcionalnosti povećava se i složenost aplikacije što otežava izvođenje izmena. Vremenom održavanje i odgovaranje na nove funkcionalne zahteve postaje dugotrajnije. Takođe greške pri izvršavanju na nekom delu sistema na bilo kom sloju mogu uticati na dostupnost cele aplikacije. Horizontalno skaliranje moguće je jedino postavljanjem više identičnih instanci cele aplikacije iza raspoređivača mrežnog opterećenja (*eng. Load balancer*).

Mikroservisna arhitektura je nastala sa ciljem da poveća fleksibilnost i proširivost aplikacije podelom na male labavo vezane servise koji se mogu nezavisno razvijati, testirati i postavljati na server [2]. Servisi su dizajnirani tako da implementiraju minimalni broj operacija koje podržavaju jednu poslovnu funkcionalnost i imaju mogućnost međusobnog komuniciranja sa drugim servisima. Ova ideja nije nova i mnogi koncepti zasnivaju se na servisno orijentisanoj arhitekturi (SOA). U odnosu na SOA, mikroservisi su manje granularnosti i jedinica ponovne upotrebe je često grupa servisa. Pored toga, svaki od servisa može imati nezavisan sistem za skladištenje podataka. Implementacija komunikacija se zasniva uglavnom na univerzalno poznatim komunikacionim stilovima kao što su REST (*eng. Representational state transfer*), gRPC (*eng. Remote procedure call*) ili drugim protokolima, i se oni mogu razlikovati među servisima. Komunikacija se odvija uglavnom bez prenošenja stanja (*eng. Stateless*), za razliku od SOA kod koje se često stanje održava putem sesija. Takođe, servisi su dizajnirani tako da budu tolerantni na nedostupnost i kvarove, odnosno drugi servisi koji ih koriste moraju biti spremni na takve situacije i odgovoriti elegantno. Na Slici 2 prikazan je primer arhitekture mikroservisa.

Prednosti mikroservisne arhitekture u odnosu na monolitnu arhitekturu su brojni. Prvenstveno, servisi se mogu potpuno nezavisno razvijati i postavljati u produkciju, nasuprot monolitnim aplikacijama kod kojih je to moguće samo u celini. Takođe na ovaj način omogućeno je skaliranje na nivou servisa, odnosno samo delova sistema gde je to potrebno. Zbog nezavisnog postavljanja na server i isporuka servisa može biti nezavisna. Omogućena je lakša primena kontinualne integracije i veći je prostor za optimizaciju sa strane infrastrukture. Ipak povećana kompleksnost operacionih zadataka prilikom razvoja, postavljanja u produkciju i nadgledanja mikroservisnih aplikacija često se spominje kao jedan od nedostataka ovog pristupa.

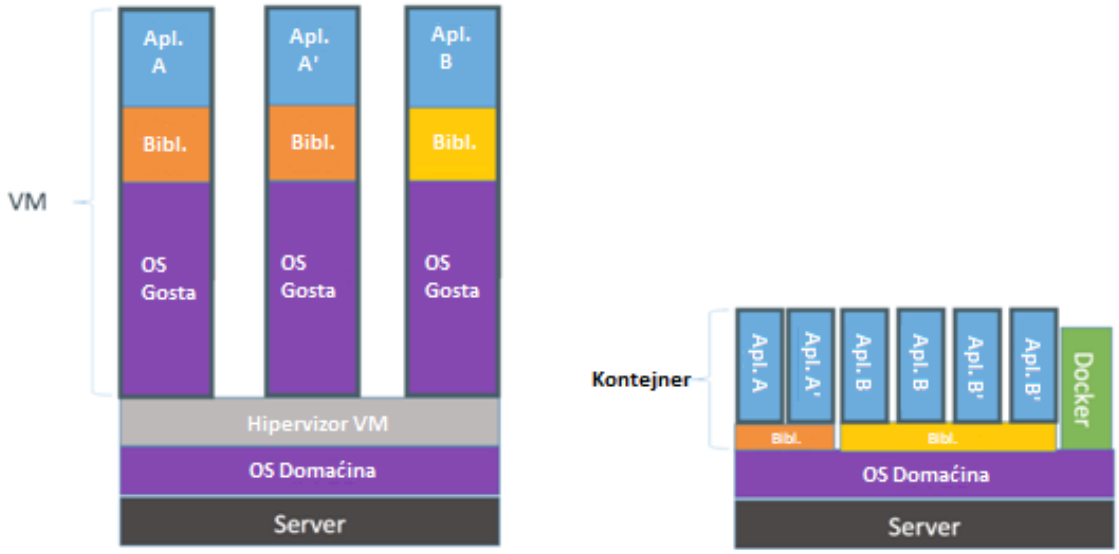


Slika 2. Mikroservisna arhitektura

2.2 Virtualizacija zasnovana na kontejnerima

Softverski kontejneri su jedan od oblika virtualizacije na nivou operativnog sistema. U idealnom slučaju slika kontejnera (*eng. Container image*) enkapsulira jedan ili više procesa zajedno sa njihovim kodom, zavisnim bibliotekama, parametrima okruženja (*eng. Environment variables*) i drugim datotekama koje su potrebne prilikom njihovog izvršavanja. Kontejnerom nazivamo jednu insancu slike koju je moguće nezavisno izvršavati. U poređenju sa virtualnim mašinama koje pokreće hipervizor (*eng. Virtual Machine Hypervisor*) i u sebi sadrže kompletan gostujući operativni sistem, softverski kontejneri koriste kernel operativnog sistema domaćina. Na operativnim sistemima Linuks izolacija između kontejnera postiže se korišćenjem prostora imena (*eng. Linux namespaces*). Na ovaj način resursi kao što su pristup sistemu datoteka, alocirana memorija i mrežni portovi jednog kontejnera su izolovani od drugih. Na Slici 3 prikazan je odnos kontejnera i virtualne mašine.

Jedna od najkorišćenijih platformi za rad sa kontejnerima je *Docker*, sa repozitorijumom *DockerHub* na kome su javno dostupne mnogobrojne čaure kontejnera kreirane od strane korisnika. Nove slike kontejnera definišu se preko komandi u posebnoj datoteci sa nazivom *Dockerfile*. Za pokretanje i upravljanje aplikacijama koje se sastoje od više kontejnera može se koristiti alat *Docker Compose*.



Slika 3. Odnos kontejnera i virtualne mašine

Orkestracija kontejnera je naziv za automatizovanu konfiguraciju, postavljanje, upravljanje i skaliranje sistema koji su zasnovani na kontejnerima. Kroz alate za orkestraciju moguće je definisati upotrebu serverskih resursa od strane kontejnera, mrežnu komunikaciju između samih kontejnera kao i sa eksternim činiocima. Pored toga omogućavaju pokretanje odnosno zaustavljanje instanci kontejnera kako bi se osiguralo željeno stanje sistema i nivo skaliranja. Neki od sistema ovog tipa su *Kubernetes*, *Docker Swarm* i drugi.

Kontejneri se često u praksi koriste sa mikroservisnom arhitekturom, na način gde se svaki mikroservis enkapsulira u posebnu kapsulu kontejnera. Neke od prednosti sistema koji koriste ovaj oblik virtualizacije su povećana enkapsuliranost i lakša prenosivost. Takođe, zbog svoje male veličine i dobre izolovanosti, kontejneri su doneli mogućnost pokretanja novih instanci servisa za veoma kratko vreme, reda veličine nekoliko sekundi. Zajedno sa alatima za orkestraciju doneli su veliku fleksibilnost u proces skaliranja sistema. Mnoge moderne platforme za računarstvo “bez servera” u pozadini koriste ovaj oblik virtualizacije.

2.3 Računarstvo u oblaku, modeli servisa

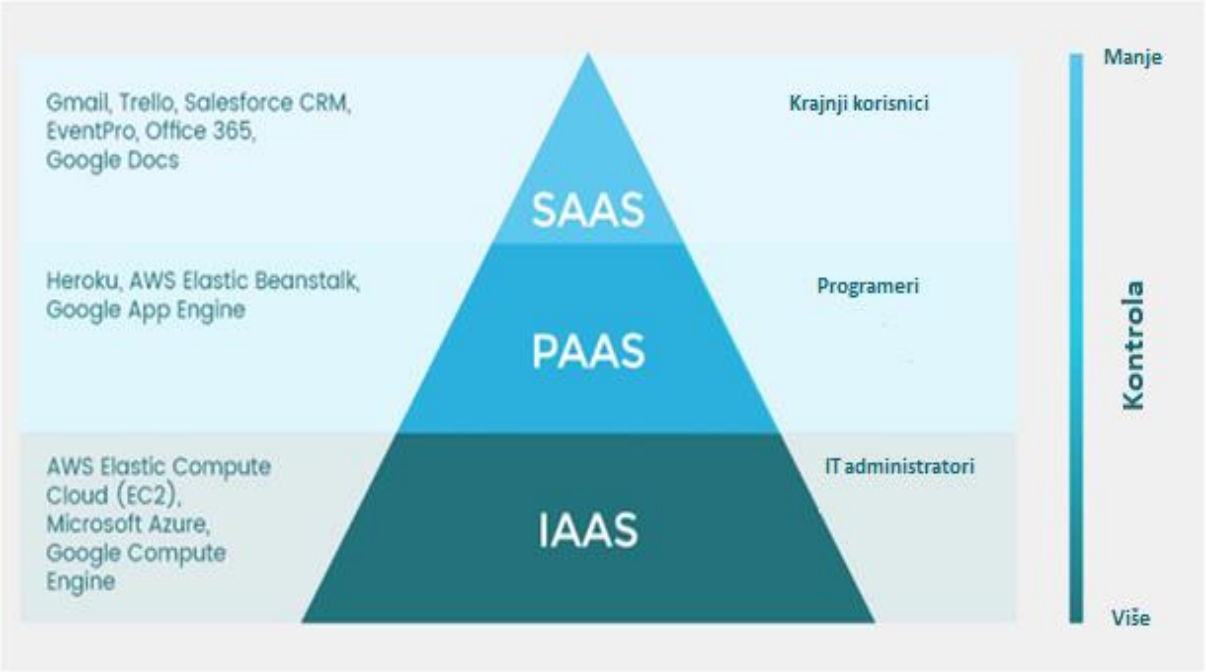
Računarstvo u oblaku u svojim počecima predstavljalo je iznajmljivanje infrastrukturnih resursa kao što su serveri ili sistemi za skladištenje podataka, tako da su korisniku dostupni na zahtev i naplativi po utrošku ili vremenu dostupnosti. Programeru je omogućeno da upravlja, održava i nadgleda svoje resurse u oblaku putem odgovarajućih servisa. Vremenom i porastom popularnosti isporučioci su uvrstili u svoje ponude veliki broj servisa različitih namena. Prema nivou raspodele odgovornosti prilikom upravljanja resursima između isporučilaca i korisnika, mogu se uočiti različiti modeli servisa [3]. Tri glavna modela servisa računarstva u oblaku su opisana u nastavku.

- "Infrastruktura kao servis" (*eng. Infrastructure as a service*) se odnosi na rezervisanje korišćenja tri vrste infrastrukturnih resursa putem servisa. To su serveri, resursi za skladištenje podataka i mrežna infrastruktura. Jedinica rezervisanja servera najčešće je virtualna mašina sa željenim operativnim sistemom i definisanim hardverskim resursima kao što su procesor, količina radne memorije i drugi. Kod servisa za skladištenje podataka moguće je rezervisati skladištenje generičkih objekata (*eng. Blob Objects*) ili specijalizovanih sistema za blokovsko skladištenje i skladištenje datoteka. Mrežni servisi omogućavaju povezivanje rezervisanih resursa u virtualnu mrežu na platformi u oblaku kao i njihovo povezivanje sa drugim mrežama ili Internetom. Plaćanje kod servisa koji pripadaju ovom modelu obavlja se najčešće po vremenu rezervisanosti, ali se može i razlikovati u zavisnosti od tipa servisa. Isporučioci na platformi uglavnom imaju u ponudi i servise koji olakšavaju operacione aktivnosti na rezervisanim infrastururnim resursima, od servisa za nadgledanje, upravljanje datotekama dnevnika, pravljenja rezervnih kopija (*eng. Backup*), bezbednosti, oporavka od havarije (*eng. Disaster recovery*) i drugih.

- "Platforma kao servis" (*eng. Platform as a service*) se odnosi na rezervisanje korišćenja platforme za razvoj, postavljanje i izvršavanje aplikacija na platformi u oblaku. Ovakve platforme nude mogućnost izbora programskih jezika, razvojnih okvira, biblioteka, korišćenja kontejnera za virtualizaciju, kontinualne integracije i drugih alata prilikom razvoja aplikacija. Osim toga omogućavaju podešavanja platforme prilikom izvršavanja aplikacije korišćenjem parametara okruženja, nadgledanje, upravljanje datotekama dnevnika i podešavanja bezbednosti na nivou aplikacije. Programer nema mogućnost direktne kontrole i operacionih aktivnosti nad infrastrukturnim resursima koji se nalaze ispod platforme. Naplata kod ovakvih servisa može

se razlikovati od vremena korišćenja, broja korisnika, količine prostora za skladištenje i drugih.

- "Softver kao servis" (*eng. Software as a service*) predstavlja iznajmljivanje korišćenja gotovih aplikacija razvijenih od strane isporučilaca ili trećih kompanija koje se izvršavaju na platformi u oblaku. Ove aplikacije najčešće se koriste preko korisničkog interfesa, dostupnog preko veb pregledača, mobilnih uređaja ili drugih. Korisnik nema kontrolu nad okruženjem aplikacije, kao ni infrastrukturnim resursima prilikom njenog izvršavanja. Upravljanje je moguće jedino u samoj aplikaciji i to nad dodeljenim korisničkim nalozima i dozvolama, odnosno limitiranim skupom funkcionalnosti same aplikacije. Plaćanje kod ovog modela se često definiše pretplatom na mesečnom ili godišnjem nivou.



Slika 4. Modeli servisa računarstva u oblaku

2.4 Računarstvo "bez servera"

Iako izraz računarstvo "bez servera" (*eng. Serverless computing*) nagoveštava da ne postoji serverska komponenta u smislu hardvera i serverskih procesa kao dela arhitekture sistema, to nije u potpunosti tačno. Zapravo odnosi se na prebacivanje odgovornosti za upravljanje serverima i drugim resursima potrebnim za izvršavanje koda na treće lice. U velikom broju primera je u pitanju isporučilac platforme u oblaku, ali to ne mora uvek biti slučaj.

U osnovi računarstvo "bez servera" se može podeliti u dve grupe servisa. Prva grupa je takozvani "zadnji kraj kao servis" (*eng. Back end as a service*). To su servisi koji su u potpunosti razvijeni od strane trećih lica i hostovani na nekoj od platformi u oblaku. Kao takve moguće ih je integrisati u veb ili mobilnu aplikaciju programera. Ovde spadaju servisi različitih namena od autentikacije korisnika (*Auth0*, *Okta*), preko baza podataka (*AWS Aurora*, *Firebase*) do servisa za slanje notifikacija i mnogih drugih. Drugu i značajniju grupu čini model "funkcija kao servis" (*eng. Function as a service*) i on će biti detaljnije opisan u narednim sekcijama.

2.4.1 Model "funkcija kao servis"

"Funkcija kao servis" je relativno nov model servisa računarstva u oblaku koji je širu popularnost stekao predstavljanjem platforme *AWS Lambda* zasnovane na ovom modelu krajem 2014 godine. Model je organizovan tako da kod na serverskoj strani piše programer u obliku funkcija koje su bez stanja, pokreću se na osnovu događaja i njihovo izvršavanje u potpunosti kontroliše isporučilac platforme.

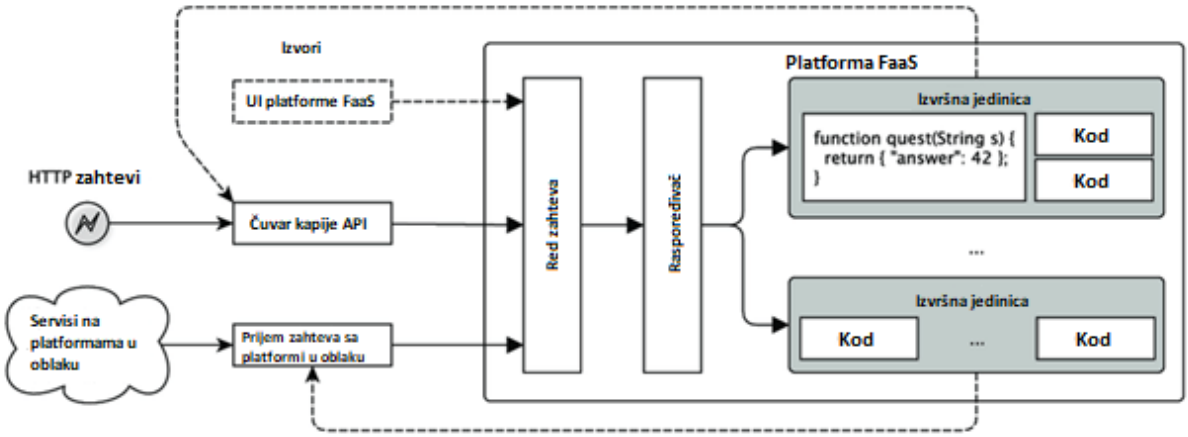
Platforme "funkcija kao servis" omogućavaju izvršavanje koda bez podešavanja servera ili na njima postavljenih serverskih procesa koji su dugog životnog veka. Kod je organizovan u obliku funkcija koje se grupišu u izvršne jedinice, često kontejnere, o čijem se pokretanju, životnom veku i upravljanju prilikom izvršavanja stara platforma. Ovde se ogleda i razlika u odnosu na model "platforma kao servis", kod koga programer nema kotrolu upravljanja serverom, ali ima nad serversim procesima putem odgovarajućih servisa [4].

Posmatrano sa strane koda, funkcije na ovom modelu su regularne i nisu zavisne od konkretnog programskog jezika ili razvojnog okvira. Isporučioci platformi u oblaku omogućavaju pisanje funkcija u većini popularnih programskih jezika kao što su *Java*, *C#*, *JavaScript*, *Python* i drugi. Sa druge strane postavljanje u produkciju

se dosta razlikuje od ostalih modela, na taj način što se kod podiže na posebno određene lokacije na plaformi isporučioca koje zatim platforma koristi za instanciranje izvršnih jedinica. Rezervisanje infrastrukturnih resursa za izvršavanje i upravljanje procesima obavlja platforma na automatizovan način.

Funkcije se pokreću putem događaja koji mogu biti HTTP zahtev, tajmer, ili događaj koji se dogodio na nekim od drugih servisa na platformi u oblaku. Veliki isporučioci omogućavaju veliki broj različitih tipova događaja na osnovu ostalih servisa koje imaju u ponudi. Primeri bi bili upis u neku od baza podataka na platformi, pristigla poruka u redu za obradu i drugi.

Na Slici 5 prikazan je uprošćen diagram arhitekture platforme na modelu "funkcija kao servis". Nakon pristizanja novog zahteva, zadatak platforme je da zahtev primi u red i zatim ga prosledi odgovarajućoj aktivnoj izvršnoj jedinici ili alocira novu izvršnu jedinicu za njegovu obradu. Platforma mora pokrenuti dovoljan broj instanci izvršne jedinice da opsluži sve pristigle zahteve u zavisnosti od količine saobraćaja. Takođe i dealocira određenu izvršnu jedinicu ukoliko je neaktivna, odnosno ukoliko je istekao definisani vremenski period nakon poslednjeg obrađenog zahteva. Na ovaj način horizontalno skaliranje je prebačeno na plaftormu, i odvija se na automatizovan način, bez bilo kakvog podešavanja programera.



Slika 5. Arhitektura platforme zasnovane na modelu "funkcija kao servis"

2.4.2 Osobine

Postoji veći broj osobina koje karakterišu računarstvo "bez servera" i model "funkcija kao servis". Njihovo poznavanje omogućava programerima poređenje različitih platformi i bolji uvid prilikom izbora i korišćenja odgovarajuće platforme.

- Čuvanje stanja – za funkcije na ovom modelu se kaže da su bez stanja (*eng. Stateless*) i podaci se čuvaju samo u promenljivim vrednostima funkcije. Odnosno nema garancija da se stanje može čuvati između više različitih poziva funkcije, i za ove potrebe moraju se koristiti eksterni sistemi za skladištenje. Ipak, neki od isporučilaca imaju u ponudi i mehanizme koji čuvaju stanje i olakšavaju ulančane pozive i integraciju više funkcija.

- Performanse – različiti činioci i limiti utiču na performanse koda na ovom modelu, od broja konkurentnih zahteva, do maksimalne veličine memorije i procesorskih resursa za jedan poziv. Pored ovih, platforme često imaju i vremensko ograničenje trajanja jednog zahteva, nakon čega se procesiranje zahteva zaustavlja. Takođe, vreme od pristizanja zahteva do početka izvršavanja funkcije se može razlikovati u slučajevima kada se koristi postojeća izvršna jedinica, što nazivamo *topli start* (*eng. Warm start*) i kada se alocira nova, odnosno *hladni start* (*eng. Cold start*). Ova ralika može iznositi od nekoliko milisekundi do čak nekoliko sekundi.

- Naplata – naplata zavisi od količine i vremenske raspoređenosti zahteva i vrši se samo za resurse utrošene prilikom izvršavanja. Sve veće platforme imaju mogućnost skaliranja do nule u periodima kada nema zahteva za obradu i u tim trenucima programer nema troškova.

- Otvorenost koda – iako je većina platformi zatvorenog koda, postoje i one otvorenog koda kao što su *OpenFaaS* i *Fission*. Ove platforme je moguće hostovati i na lokalnim serverima na kojima postoji *Kubernetes* ili drugi odgovarajući sistem za orkestraciju kontejnera. Od velikih javnih isporučilaca platforme koje su otvorenog koda su *IBM Cloud Functions* koji je baziran na platformi *Apache OpenWhisk* i *Microsoft Azure Functions*, i njihovi kodovi su javno dostupni na servisu *GitHub* pod licencom *Apache*, odnosno *MIT*.

- Bezbednost – bezbednost na ovom modelu ima dosta sličnosti sa modelom platforma kao servis. S obzirom da nema servera ili virtualnih mašina, nije potrebno održavanje sigursnosnih zakrpa na tom nivou. Isporučioci platformi u oblaku često imaju posebne servise za upravljanje nalozima i

dozvolama na platformi poput *AWS IAM* ili *Azure AD* i oni imaju primenu i ovde.

- Lokalni razvoj, nadgledanje i testiranje – neki od isporučilaca imaju alate koji olakšavaju lokalno pokretanje i debagovanje funkcija. Kod drugih debagovanje i nadgledanje je moguće kroz upis u dnevnik datoteke funkcija. Jedinično testiranje funkcija je jednostavno s obzirom da se one sastoje samo od koda. Sa druge strane, integraciono testiranje složenih aplikacija na ovom modelu može biti veliki izazov.

2.4.3 Prednosti i nedostaci

Neke od prednosti ovog modela proizilaze direktno iz njegovog načina rada. Zbog automatskog skaliranja i ne korišćenja resursa u neaktivnom stanju, programer može imati veće uštede prilikom naplate servisa. Najbolji primeri su aplikacije kod kojih postoje periodi kada nema zahteva ili je saobraćaj nekonzistentan. Takođe, u odnosu na sisteme zasnovane na virtualnim mašinama i kontejnerima, može se govoriti o uštedi u operacionim aktivnostima, zbog toga što upravljanje infrastrukturom obavlja isporučilac. Prednost ovog modela kod razvoja kompletno novih aplikacija je i to što se programer može više fokusirati na sam dizajn arhitekture i aplikativni kod, i tako brže doći do prvih upotrebljivih verzija aplikacije.

Sa druge strane, zbog vremenskog ograničenja za izvršavanje zahteva i odsustva čuvanja stanja na serverskoj strani, arhitektura "bez servera" nije adekvatna za neke vrste aplikacija. Slično se može reći i u slučajevima u kojima postoji potreba programera za specifičnom kontrolom i konfiguracijom infrastrukture. Generalno, aplikacije na ovom modelu su manje prenosive u odnosu na standardne virtualne mašine ili kontejnere. Događaji koji prouzrokuju pokretanje funkcija mogu se bazirati na drugim servisima platforme u oblaku, što može dovesti do veće zavisnosti programera od konkretnog isporučioca platforme.

3 Platforme “funkcija kao servis”

Jedna od prvih i najpoznatijih platformi na ovom modelu je svakako *Amazon AWS Lambda*. Ova platforma omogućava pisanje funkcija bez stanja u jezicima *Java*, *Python*, *C#*, *Node.js*, *Go*, *Ruby* i *PowerShell* [5]. Funkcije se pokreću na osnovu događaja na drugim servisima na *AWS* platformi u oblaku, kao što su postavljanje dokumenta na servis *S3* za skladištenje, prosleđen HTTP zahtev sa servisa *API Gateway* i mnogi drugi. Za debagovanje i nadgledanje aplikacija mogu se koristiti *AWS X-Ray* i *Amazon Cloud Watch* servisi. Pored toga dostupan je *AWS Serverless Application Repository* gde je moguće pronaći gotove aplikacije ili funkcije razvijene od strane zajednice koje je moguće koristiti.

Google Cloud Functions je “funkcija kao servis” platforma kompanije *Google*. Podržani jezici su *Go*, *Node.js*, *Java* i *Python* [5]. Platforma ima integrisane funkcionalnosti za debagovanje, nadgledanje i upis u dnevnik datoteke. Funkcije se pokreću na osnovu HTTP zahteva ili događaja na servisima platforme *Google Cloud*. Kod funkcija je moguće modifikovati lokalno i preko veb portala platforme. Platforme velikih ispručioca na ovom modelu upotpunjuju *IBM Cloud Functions* i *Oracle Cloud Functions*.

Jedan od najznačajnijih platformi “funkcija kao servis” koja spada u softver otvorenog koda je *Apache OpenWhisk*. Platfomu je moguće hostovati na *Kubernetes* klasteru na lokalnim serverima ili na nekim od *Kubernetes* servisa na platformama u oblaku. Takođe *IBM Cloud Functions* je bazirana na ovom projektu i moguće je koristiti kao gotovo rešenje na platformi ovog isporučioca. Podržani su jezici *Go*, *Java*, *JavaScript*, *C#*, *Python*, *PHP*, *Ruby* i *Swift*. Projekat pruža i alate komandne linije za kreiranje i debagovanje funkcija, kao i za postavljanje i upravljanje funkcijama u produkciji. Na sličan način funkcionišu i druge platforme otvorenog koda, kao što su *OpenFaas*, *Knative* i *Kubeless*.

3.1 Azure Functions

Azure Functions je platforma na modelu "funkcija kao servis" kompanije *Microsoft* i deo je *Microsoft Azure* platforme za računarstvo u oblaku. Zajedno sa servisima *Logic Apps*, *Event Grid* i *CosmosDb* čini grupu servisa koji omogućavaju računarstvo „bez servera" na ovoj platformi. Razvijena je kao softver otvorenog koda i prvi put predstavljena u januaru 2017. godine. Pored *AWS Lambda* i *Google Functions* spada u tri najpopularnije platforme na ovom modelu. U narednom delu dat je detaljniji pregled ove platforme sa praktičnim primerima.

3.1.1 Osobine platforme, programski jezici i planovi korišćenja

Platforma *Azure Functions* je zadužena za izvršavanje funkcija na plaftormi u oblaku, a alternativno moguće je i njeno hostovanje na lokalnim serverima. U trenutku pisanja ovog rada aktuelna je bila stabilna verzija platforme 3.1. Za razvoj funcija inicijalno bili su podržani jezici *C#*, *JavaScript* i *F#*, a kasnije verzije donele su podršku za druge jezike. U Tabeli 1 prikazana je podrška jezika i njihovih radnih okvira po verzijama, a postoji mogućnost dodavanja podrške za nove jezike korišćenjem jezičkih proširenja (*eng. Language Extensibility*).

Tabela 1. Podržani jezici

Jezik	**1.x**	**2.x**	**3.x**
C#	Da (*.NET* 4.7)	Da (*.NET Core* 2.2)	Da (*.NET Core* 3.1)
JavaScript	Da (*Node* 6)	Da (*Node* 10 i 8)	Da (*Node* 12 i 11)
F#	Da (*.NET* 4.7)	Da (*.NET Core* 2.2)	Da (*.NET Core* 3.1)
Java	Ne	Da (*Java* 8)	Da (*Java* 11 i 8)
PowerShell	Ne	Da (*PowerShell* 6)	Da (*PowerShell* 7 i 6)
Python	Ne	Da (*Python* 3.7 i 3.6)	Da (*Python* 3.8, 3.7 i 3.6)
TypeScript	Ne	Da	Da

Kao i kod većine ostalih servisa na platformi u oblaku *Microsoft Azure*, platforma je dostupna kao servis i oslanja se na planove za korišćenje (*eng. Azure App Service plans*). Različiti planovi korišćenja definišu resurse koje je moguće koristiti, region

dostupnosti, načine hostovanja, dinamiku skaliranja, limite u veličini zahteva, maksimalnu količinu memorije, veličinu prostora na nalogu za skladištenje, maksimalno vreme izvršavanja (*eng. Execution timeout*) i druge parametre prilikom hostovanja [6]. Ponuđeno je pet planova za korišćenje platforme *Azure Functions* korisnicima i oni su redom: *Consumption plan*, *Premium plan*, *Dedicated plan*, *ASE* i *Kubernetes*.

3.1.2 Funkcije i projekti funkcija

Validne funkcije se sastoje od dve datoteke, prve koja sadrži kod za izvršavanje u jeziku koji je korisnik odabrao i druge pod nazivom **`function.json`** koja sadrži konfiguracioni kod u formatu JSON. U ovoj datoteci definisan je okidač, sva vezivanja i dodatni konfiguracioni parametri okruženja za tu funkciju. Za kompilirane programske jezike moguće je automatski generisati datoteku **`function.json`** prilikom faze kompilacije, dok se za interpretirane jezike ona mora posebno napisati.

Funkcije su najčešće grupisane u projekte funkcija (*eng. Function App*). Projekti omogućavaju lakše upravljanje grupom funkcija i njihovo postavljanje i podešavanje na plaftormi *Microsoft Azure*. Od platforme verzije 2.0, funkcije koje su deo istog projekta moraju biti napisane u istom programskom jeziku i koristiti istu verziju okruženja. Projekti imaju definisanu strukturu direktorijuma kako bi postavljanje na platformu i izvršavanje bilo uniformno i ona mora biti poštovana bez obzira na programski jezik i radni okvir razvoja [7].

```
Projekat
 | - host.json
 | - PrvaFunkcija
 | | - function.json
 | | - ...
 | - DrugaFunkcija
 | | - function.json
 | | - ...
 | - DeljeniKod
 | - bin
```

Slika 6. Struktura projekta funkcija

Na Slici 6 data je organizacija tipičnog projekta funkcija. U čvornom direktorijumu se nalazi datoteka **`host.json`** sa konfiguracionim podešavanjima. Svaka od funkcija smeštena je u posebnom poddirektorijumu, kao i opcioni deljeni kod, dok se u bin direktorijumu nalaze izvršne datoteke. U zavisnosti od programskog jezika mogu biti definisana dodatna pravila u strukturi.

3.1.3 Lokalni razvoj

Za potrebe pokretanja funkcija na lokalnom računaru potrebno je instalirati alat komandne linije *Azure Functions Core Tools* [8] kao i radni okvir za programski jezik koji je izabran, njihove trenutne verzije su prikazane u Tabeli 1. Alternativno za pokretanje funkcija mogu se koristiti dodataci za razvojno okruženje *Visual Studio* i *Visual Studio Code*, a za pisanje koda samih funkcija i druga okruženja koja olakšavaju rad u izabranom jeziku.

U nastavku biće dat primer u jeziku *C#* i korišćenjem *Azure Functions Core Tools* [8] alata iz komandne linije. Na Primeru koda 1 data je komanda kojom kreira se biblioteka klasa (*eng. Class library*) koja predstavlja projekat funkcija.

Primer koda 1. Kreiranje projekta

```
func init Aplikacija --dotnet
```

Kod funkcija nalazi se u datotekama sa ekstenzijom „.cs“ u odgovarajućim direktorijumima. Na Primeru koda 2 dat je primer koda funkcije sa okidačem na HTTP zahtev. Funkcija čita sadržaj parametra ime i na osnovu njega vraća odgovarajuću poruku u telu HTTP odgovora. Iz primera se može videti da se okidači mogu specifikovati kao *C#* atributi parametara funkcije. Više reči o okidačima i vezivanjima biće u narednoj sekciji.

Primer koda 2. Primer funkcije sa Http okidačem

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace Aplikacija
{
    public static class Funkcije
    {
        [FunctionName("HttpPrimer")]
        public static async Task<IActionResult> HttpPrimer(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get")] HttpRequest req,
            ILogger log)
        {
            string ime = req.Query["ime"];

            log.LogInformation($"Primer funkcija je obradila zahtev - Ime: {ime}.");

            return ime != null
                ? (ActionResult)new OkObjectResult($"Zdravo, {ime}!")
                : new BadRequestObjectResult("Nije prosledjen parametar ime.");
        }
    }
}
```

Kompiliranje projekta sa funkcijama se vrši komandom datom na Primeru koda 3. Nakon toga funkcija je dostupna lokalno za pozivanje na predefinisanom portu 7071, odnosno na lokaciji http://localhost:7071/api/httpPrimer.

Primer koda 3. Kompilacija i pokretanje

```
func start --build
```

Na ovaj način automatski je izgenerisana i konfiguraciona datoteka **`function.json`** i njen sadržaj dat je na Primeru koda 4.

Primer koda 4. Primer function.json datoteke

```
{
  "generatedBy": "Microsoft.NET.Sdk.Functions-1.0.24",
  "configurationSource": "attributes",
  "bindings": [
    {
      "type": "httpTrigger",
      "methods": [
        "get"
      ],
      "authLevel": "anonymous",
      "name": "req"
    }
  ],
  "disabled": false,
  "scriptFile": "../bin/Aplikacija.dll",
  "entryPoint": "Aplikacija.Funkcije.HttpPrimer"
}
```

3.1.4 Okidači i vezivanja

Vezivanja (*eng. Bindings*) definišu načine na koji funkcija komunicira sa spoljašnjim svetom ili ostalim servisima programera na platformi *Microsoft Azure* [7]. Funkcija može imati veći broj vezivanja i ona mogu biti ulazna, izlazna ili dvosmerna. Podaci iz ulaznih vezivanja su prilikom izvršavanja dostupni kao parametri funkcije, dok se na izlazna vezivanja mogu slati podaci u telu funkcije ili kao njena povratna vrednost. Primeri ulaznog vezivanja bi bili tajmer, HTTP zahtev, upis koji se dogodio na servisu *Blob storage*, ulazna poruka na servisu *Queue Storage*, događaj u bazi podataka na servisu *CosmosDb* ili na drugim servisima. Izlazno vezivanje može biti prosleđivanje rezultata funkcije na ove ili druge servise. U tabeli 2 data su svi podržani tipovi vezivanja.

U datoteci **`function.json`** vezivanja su definisana u posebnom nizu sa istim nazivom (*eng. Bindings*). Svaki element niza minimalno sadrži parametre tip (*eng. Type*) servisa za koji se definiše vezivanje, naziv (*eng. Name*), smer (*eng. Direction*) i tip podataka (*eng. DataType*) koje vezivanje očekuje. Ukoliko se koristi jezik *C#* moguće je specifikovanje vezivanja preko strogo tipiziranih *C#* atributa u kodu funkcije, da bi se na osnovu njih u fazi kompilacije generisala odgovarajuća sekcija u datoteci **`function.json`**.

Tabela 2. Podržani okidači i vezivanja

Servis	1.x	2.x i više	Okidač	Ulazno vezivanje	Izlazno vezivanje
Blob storage	✓	✓	✓	✓	✓
CosmosDb	✓	✓	✓	✓	✓
Dapr		✓	✓	✓	✓
Event grid	✓	✓	✓		✓
Event hubs	✓	✓	✓		✓
HTTP	✓	✓	✓		✓
IoT hubs	✓	✓	✓		✓
Kafka		✓	✓		✓
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
RabbitMQ		✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓		✓	✓
Table storage	✓	✓		✓	✓
Tajmer	✓	✓	✓		
Twillio	✓	✓			✓

Poseban tip ulaznog vezivanja koja prouzrokuje izvršavanje funkcije je okidač i svaka funkcija mora imati tačno jedan okidač. Okidači u svom nazivu moraju imati nastavak “Trigger“ kako bi se razlikovali od ostalih ulaznih vezivanja. Na Primeru koda 5 dat je primer na kome je okidač poruka sa servisa *Queue Storage*. Poruka se zatim formatira i prosleđuje na izlazno vezivanje što je u ovom slučaju nova datoteka na servisu *Blob storage*. Na Primeru koda 6 je data odgovarajuća datoteka `function.json` za ovu funkciju.

Primer koda 5. Primer funkcije sa ulaznim i izlaznim vezivanjem

```
        [FunctionName("VezivanjaPrimer")]
        public static async Task VezivanjaPrimer(
            [QueueTrigger("osobe-red")] Osoba osoba,
            [Blob("osobe/{rand-guid}.json")] TextWriter blobWriter,
            ILogger log)
        {
            string json = string.Format(
                "{{ \"id\": \"{0}\", \"ime\": \"{1}\" }}",
                osoba.Id,
                osoba.Ime);

            await blobWriter.WriteAsync(json);

            log.LogInformation($"Obradjena je poruka iz reda osoba. Osoba: {osoba.Id}.");
        }
```

Primer koda 6. Primer konfiguracije vezivanja

```
{
  "generatedBy": "Microsoft.NET.Sdk.Functions-1.0.24",
  "configurationSource": "attributes",
  "bindings": [
    {
      "type": "queueTrigger",
      "queueName": "osobe-red",
      "direction": "in",
      "name": "osoba"
    },
       {
      "type": "blob",
      "path": "osobe",
      "direction": "out",
      "name": "blobWritter"
    }
  ],
  "disabled": false,
  "scriptFile": "../bin/Aplikacija.dll",
  "entryPoint": "Aplikacija.Funkcije.VezivanjaPrimer"
}
```

3.1.5 Trajne funkcije

Trajne funkcije (*eng. Durable functions*) je nadogradnja na platformu *Azure Functions* koja je otvorenog koda i omogućava pisanje funkcija koje imaju mogućnost čuvanja stanja. To postižu uvođenjem novih tipova funkcija:

- Funkcije orkestratori (*eng. Orchestrator functions*) – su funkcije koje imaju samo jednu odgovornost, i to je organizacija toka izvršavanja, dok sve ostale

zadatke delegiraju funkcijama aktivnosti. Tokom izvršavanja aktivnosti orkestrator funkcija je u neaktivnom stanju, u trenutku kada se poziv završi okrestrator funkcija nastavlja sa radom i poziva sledeću aktivnost.

- Funkcije aktivnosti (*eng. Activity functions*) – su regularne funkcije koje imaju dodatan parametar u vezivanju tipa `IDurableOrchestrationContext` preko kojeg mogu komunicirati sa funkcijama orkestratorima. Preko ovog parametra one mogu primati ulazne informacije i takođe vraćati podatke orkestratoru.

3.1.6 Postavljanje na Azure platformu

Za postavljanje lokalno razvijenog projekta sa funkcijama na platformu potrebno je da prethodno budu kreirani zavisni resursi na platformi i to resursna grupa (*eng. Resource group*), nalog za skladištenje (*eng. Storage account*), a potom i sam projekat sa funkcijama. Kreiranje resursa je moguće uraditi na više načina, preko veb portala platforme, korišćenjem alata komandne linije (*eng. Azure CLI*) ili korišćenjem *ARM* šablona (*eng. ARM template*). Na Primeru koda 7 prikazano je rezervisanje novog projekta funkcija na platformi u oblaku iz komandne linije.

Primer koda 7. Rezervisanje novog projekta funkcija na platformi

```
az functionapp create --resource-group res-grupa-rg --consumption-plan-location
germanycentral --runtime dotnet --functions-version 2 --name aplikacija --storage-account
sklad-nalog
```

Prilikom kreiranja moguće je specifikovati region, razvojnu platformu, verziju izvršnog okruženja, naziv aplikacije, naziv resursne grupe, nalog za skladištenje, operativni sistem, plan korišćenja, i da li će se prilikom postavljanja koristiti kod projekta ili priloženi *Docker* kontejner.

Na nalogu za skladištenje čuvaju se datoteke paketa u tri direktorijuma redom *data*, *logFiles* i *site*. U *data* direktorijumu se čuvaju `host.json` i druge datoteke za konfiguraciju izvršnog okruženja, *logFiles* čuva datoteke dnevnika koje nastaju prilikom izvršavanja, a u direktorijumu *site* se nalaze datoteke projekta funkcija po definisanoj strukturi ili *Docker* kontejner ukoliko je tako odabrano prilikom kreiranja resursa. Platforma koristi ove datoteke za pokretanje novih instanci

izvršnih jedinica projekta u slučajevima kada je to potrebno. Postavljanje nove verzije projekta omogućeno je na više načina od kojih su najkorišćenija dva:

- Postavljanje iz zip datoteke (*eng. Zip deployment*) – koristi se zip datoteka koja sadrži datoteke aplikacije nakon kompilacije, postavljanje se vrši preko odgovarajućeg alata, alata komandne linije, veb portala platforme ili HTTP zahteva.

- Pokretanje iz paketa (*eng. Run from package*) – postavljanjem parametra pod nazivom **`WEBSITE_RUN_FROM_PACKAGE`** u **`host.json`** datoteci, čija se vrednost postavi na link sa paketom za pokretanje koji je javno dostupan na internetu.

Na Primeru koda 8 je data komanda za postavljanje iz lokalne zip datoteke.

Primer koda 8. Postavljanje paketa na platformu iz zip datoteke

```
az functionapp deployment source config-zip res-grupa-rg -g -n aplikacija --src
aplikacija_v2.zip
```

Osim ovih omogućeno je i postavljanje novih verzija aplikacije na platfomu koristeći alate za kontinualnu integraciju *Azure DevOps*, *GitHub Actions*, *Jenkins* i drugih.

4 Razvoj REST servisa “Recepti API”

U ovom poglavlju predstavljena je implementacija servisa “Recepti API” koji je zasnovan na arhitekturi “bez servera” i platformi *Azure Functions* koji su opisani u prethodna dva poglavlja ovog rada. U prvom delu poglavlja je predstavljena implementacija servisa, dok je proces rezervisanja resutsa, postavljanja i podešavanja servisa na platformi prikazan u drugom delu poglavlja. Servis je postavljen i javno dostupan na *Microsoft Azure* platformi korišćenjem funkcionalnosti besplatnog naloga.

4.1 Implementacija servisa

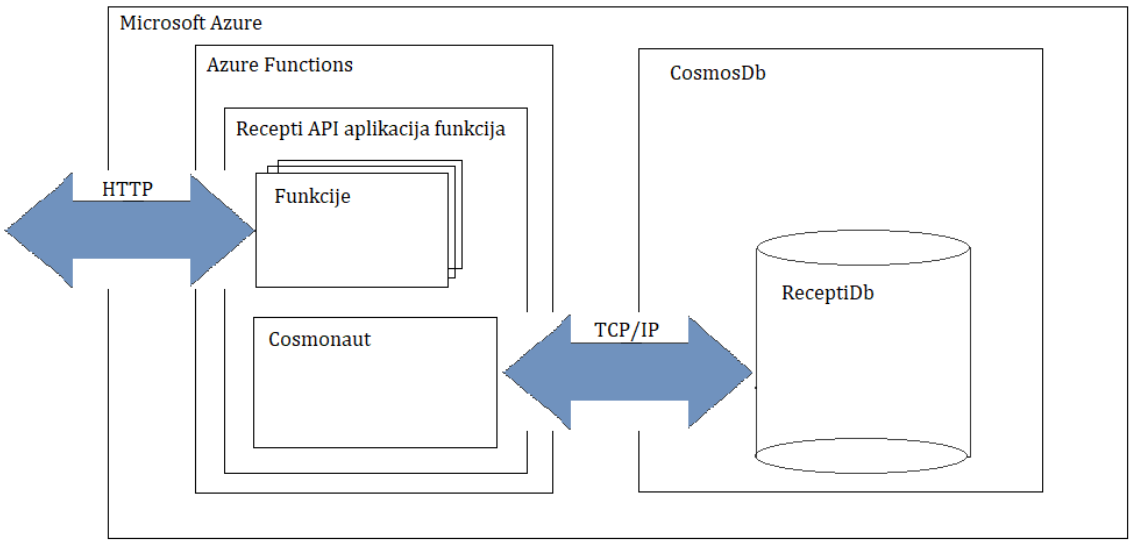
U ovom delu prikazana je implementacija servisa “Recepti API”. Servis je razvijen kao projekat otvorenog koda i celokupan kod dostupan je javno na adresi https://github.com/milosmi11166/Master. Za razvoj servisa korišćen je jezik *C#*, radni okvir *.Net Core* verzije 2.1 i razvojno okruženje *Visual Studio 2019*. U nastavku dati su funkcionalni opis i arhitektura, kao i struktura projekta i objašnjenje glavnih delova koda.

4.1.1 Funkcionalni opis i arhitektura

Servis “Recepti API” omogućava korisnicima pretragu i upravljanje kulinarskim receptima, njihovim sastojcima i koracima pripreme. Pored toga servis pruža mogućnost upravljanja namirnicama i na osnovu njih dobijanja informacija o osnovnim nutritivnim vrednostima recepata kao što su broj kalorija, količina proteina, masti, šećera i vlakana.

Arhitekturu servisa čine dve komponente. Prva komponenta se odnosi na projekat funkcija na platformi *Azure Functions*. Ova komponenta zadužena je za prihvatanje i obradu zahteva kreiranih od strane korisnika. Ukoliko za to postoji potreba, funkcije mogu komunicirati sa bazom podataka servisa. Druga komponenta je baza *ReceptiDb* i njena funkcija je skladištenje kolekcija sa podacima o receptima. Instanca baze postavljena je na servisu *CosmosDb* na platformi u oblaku.

Komunikacija funkcija sa bazom podataka implementirana je pomoću biblioteke *Cosmonaut* [9]. Na Slici 7 prikazano je kako su komponente postavljene i način na koji komuniciraju.



Slika 7. Arhitektura servisa "Recepti API"

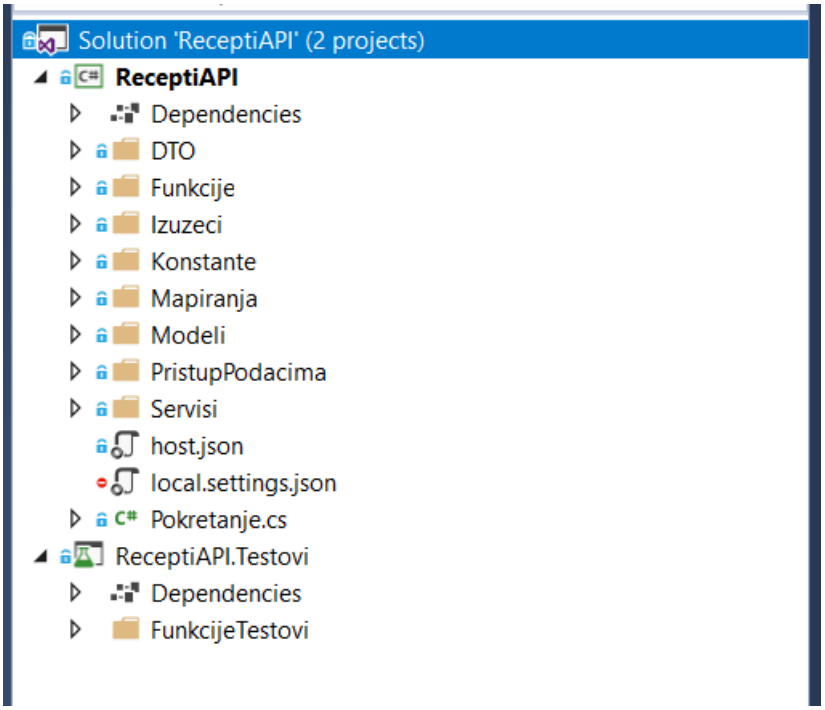
Komponenta *ReceptiApi* je zasnovana na arhitekturi "bez servera" i u trenucima kada postoji saobraćaj aktivna je jedna ili više njenih instanci na platformi. Ova komponenta sadrži servis koji zasnovan na REST arhitekturnom stilu i moguće ga je koristiti preko HTTP protokola od strane korisnika ili drugih servisa i aplikacija. U tabeli 3 prikazani su svi resursi servisa. Za autentikaciju servis koristi ugrađene ključeve koje korisnici moraju postaviti u zaglavlje *x-functions-key* svakog upućenog HTTP zahteva.

Tabela 3. Resursi servisa

Metod	Ruta	Opis
GET	/recepti?opis={o}&brojStrane={bs}&velicinaStrane={vs}	Pronalazi sve recepte po delu opisa recepta i broju strane
GET	/recepti/{id}	Pronalazi jedan recept, i vraća sve njegove sastojke, korake pripreme i nutritivne informacije
POST	/recepti	Kreira novi recept
PUT	/recepti/{id}	Ažurira recept
DELETE	/recepti/{id}	Briše recept
POST	/recepti/{idRecepta}/sastojci	Kreira sastojak za recept sa zadatim identifikatorom
PUT	/recepti/{idRecepta}/sastojci/{idNamirnice}	Ažurira sastojak
DELETE	/recepti/{idRecepta}/sastojci/{idNamirnice}	Briše sastojak
POST	/recepti/{idRecepta}/koraci-pripreme	Kreira korak pripreme za recept sa zadatim identifikatorom
PUT	/recepti/{idRecepta}/koraci-pripreme/{idKorakaPripreme}	Ažurira korak pripreme
DELETE	/recepti/{idRecepta}/koraci-pripreme/{idKorakaPripreme}	Briše korak pripreme
GET	/recepti?naziv={n}&brojStrane={bs}&velicinaStrane={vs}	Pronalazi sve namirnice po delu naziva i broju strane
GET	/namirnice/{id}	Pronalazi jednu namirnicu
POST	/namirnice	Kreira namirnicu
PUT	/namirnice/{id}	Ažurira namirnicu
DELETE	/namirnice/{id}	Briše namirnicu

4.1.2 Struktura projekta

Projekat je podeljen na dve biblioteke klasa (*eng. Class library*) u okviru rešenja za okruženje *Visual Studio*. Prvu čini projekat funkcija sa nazivom **`ReceptiAPI`** koja sadrži implementaciju funkcija i namenjena je za izvršavanje na platformi, dok je druga sa nazivom **`ReceptiAPI.Testovi`** namenjena za pisanje i izvršavanje jediničnih testova. Na Slici 8 prikazana je organizacija projekta. U nastavku opisani su direktorijumi i njihovo značenje, kao i datoteke koje se u njima nalaze.



Slika 8. Organizacija projekta

U okviru projekta **`ReceptiAPI`**:

- **`Funkcije`** – direktorijum sadrži klase sa funkcijama.
- **`DTO`** (*eng. Data Transfer Objects*) – direktorijum sadrži klase sa podacima koje funkcije koriste za komunikaciju sa eksternim svetom, serijalizovane u JSON format.
- **`Modeli`** – direktorijum sadrži klase entiteta modela za skladištenje podataka.
- **`Izuzeci`** – direktorijum sadrži izuzetke servisa.

- **`Mapiranja`** – direktorijum sadrži klase za definisanje mapiranja između DTO objekata i objekata modela.

- **`PristupPodacima`** – direktorijum sadrži klase i interfejse repozitorijuma za pristup bazi podataka.

- **`Servisi`** – direktorijum sadrži klase i interfejse servisa sa implementacijom poslovne logike.

- **`host.json`** – datoteka za čuvanje parametara podešavanja platforme *Azure Functions.*

- **`local.settings.json`** – datoteka za čuvanje parametara okruženja aplikacije (*eng. Environment Variables*) u slučajevima kada se aplikacija lokalno pokreće.

- **`Pokretanje.cs`** – klasa za imeplementaciju ponašanja prilikom inicijalnog pokretanja i registrovanje servisa.

U okviru **`ReceptiAPI.Testovi`** projekta:

- **`FunkcijeTestovi`** – direktorijum sadrži jedinične testove funkcija iz projekta **`ReceptiAPI`**.

4.1.3 Funkcije i jedinični testovi

Sve funkcije u projektu su implementirane tako da im je okidač HTTP zahtev i podatke iz zahteva i odgovora proizvode u obliku formata JSON. Na Primeru koda 9 prikazan je deo klase **`ReceptiFunkcije`** koja predstavlja funkciju **`KreirajRecept`**. Okidač je definisan preko atributa **`HttpTrigger`** preko kojeg su kao parametri postavljeni tip autorizacije, HTTP metod, ruta i klasa za deserijalizaciju tela zahteva.

Primer koda 9. Funkcija KreirajRecept

```
        [FunctionName("KreirajRecept")]
        public  async Task<JsonResult> KreirajRecept(
            [HttpTrigger(AuthorizationLevel.Function, "POST", Route = "v1/recepti")]
[FromBody] ReceptDTO receptDTO)
        {
            _dnevnik.LogInformation("KreirajRecept funkcija je primila zahtev.");

            var odgovor = new JsonResult(null);
            ReceptDTO kreiraniReceptDTO = null;

            try
            {
                kreiraniReceptDTO = await _receptiServis.Kreiraj(receptDTO);

                odgovor.StatusCode = StatusCodes.Status201Created;
                odgovor.Value = kreiraniReceptDTO;
            }
            catch (ReceptiAPIIzuzetak rai)
            {
                odgovor.StatusCode = rai.HttpStatusKod;
                odgovor.Value = new GreskaDTO { PorukaGreske = rai.Poruka };
            }
            catch (Exception i)
            {
                _dnevnik.LogError("Neobradjen izuzetak u funkciji KreirajRecept.", i);

                odgovor.StatusCode = StatusCodes.Status500InternalServerError;
                odgovor.Value = new GreskaDTO {
                     PorukaGreske = KonstantneVrednosti.GreskaNaServerskojStrani
                };
            }

            return odgovor;
        }
```

Klase sa funkcijama mogu imati zavisne klase, kao što su servisi, dnevnici, repozitorijumi i drugi. U okviru celog projekta zavisnosti su definisane kao interfejsi i umetanje konkretnih zavisnih klasa obavlja se preko konstruktora (*eng. Constructor based dependency injection*). Na ovaj način omogućena je inverzija kontrole i kreiranje jediničnih testova je značajno olakšano. Korišćen je ugrađeni kontejner inverzije kontrole iz radnog okvira *.Net Core* i njegovo ponašanje definisano je u klasi **`Pokretanje`**. Na Primeru koda 10 prikazan je deo klase **`Pokretanje`** koji definiše konkretne zavisnosti klase koje se koriste u funkciji **`KreirajRecept`** koja je prethodno prikazana.

Primer koda 10. Klasa Pokretanje

```
[assembly: WebJobsStartup(typeof(Pokretanje))]
namespace ReceptiAPI
{
    public class Pokretanje : IWebJobsStartup
    {
        public void Configure(IWebJobsBuilder graditelj)
        {
            . . .


            graditelj.Services.AddLogging();
            . . .

            graditelj.Services.AddScoped<IReceptiServis, ReceptiServis>();

            . . .
        }
    }
}
```

Jedinični testovi implementirani su u posebnom projektu pod nazivom **`ReceptiAPI.Testovi`**. Za kreiranje lažnih zavisnih objekata (*eng. Mock objects*) korišćena je biblioteka *Moq*, a za kreiranje i pokretanje jediničnih testova biblioteka *NUnit*. Na Primeru koda 11 prikazan je primer jediničnog testa za funkciju **`KreirajRecept`**.

Primer koda 11. Jedinični test funkcije KreirajRecept

```
        [Test]
        public async Task KreirajRecept_SaUspesnimUpisom_TrebaDaVratiUspesanOdgovor()
        {
            //Podesi
            ReceptDTO recept = new ReceptDTO
            {
                Id = "123",
                Naziv = "Pita sa jabukama",
                Opis = "Opis"
            };

            _receptiServisMok.Setup(x => x.Kreiraj(It.IsAny<ReceptDTO>()))
                .ReturnsAsync(recept);

            //Izvrsi
            var odgovor = await _receptiFunkcije.KreirajRecept(recept);
            ReceptDTO odgovorDTO = (ReceptDTO)odgovor.Value;

            //Potvrdi
            Assert.AreEqual(201, odgovor.StatusCode);
            Assert.AreEqual("123", odgovorDTO.Id);
            Assert.AreEqual("Pita sa jabukama", odgovorDTO.Naziv);
            Assert.AreEqual("Opis", odgovorDTO.Opis);
        }
```

4.1.4 Baza i model podataka

Sve funkcije u projektu su bez stanja i za trajno skladištenje podataka o receptima servis koristi *Azure CosmosDb* bazu podataka. *CosmosDb* je globalno distribuirana nerelaciona baza koja omogućava lako horizontalno skaliranje, više različitih modela i može se koristiti "bez servera". Za potrebe projekta kreirana je dokumentno-orjentisana baza podataka na modelu *MongoDb* koja podatke čuva u kolekcijama u JSON formatu i za upite koristi prilagođen SQL dijalekat. Detaljan prikaz rezervisanja baze podataka i drugih resursa na platformi *Azure* biće dat u drugom delu ovog poglavlja.

Za pristup bazi podataka korišćena je biblioteka *Cosmonaut* [9]. Pristup je enkapsuliran u klasi **`Repozitorijum`** koja sadrži metode za operacije kreiranja, čitanja, ažuriranja i brisanja (*eng. CRUD*) i druge upite nad bazom. Biblioteka *Cosmonaut* omogućava čuvanje više entiteta modela uokviru iste kolekcije u bazi podataka preko interfejsa **`ISharedCosmosEntity`**. Svaki entitet modela ima odgovarajuću klasu u kojoj je definisana njegova struktura. Klasa entiteta **`Namirnica`** data je na Primeru koda 12.

Primer koda 12. Klasa entiteta Namirnica

```
[SharedCosmosCollection("Recepti")]
public class Namirnica : ISharedCosmosEntity
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }
    [JsonProperty(PropertyName = "naziv")]
    public string Naziv { get; set; }
    [JsonProperty(PropertyName = "opis")]
    public string Opis { get; set; }
    [JsonProperty(PropertyName = "kategorija")]
    public string Kategorija { get; set; }
    [JsonProperty(PropertyName = "kalorije")]
    public decimal Kalorije { get; set; }
    [JsonProperty(PropertyName = "proteini")]
    public decimal Proteini { get; set; }
    [JsonProperty(PropertyName = "masti")]
    public decimal Masti { get; set; }
    [JsonProperty(PropertyName = "zasiceneMasti")]
    public decimal ZasiceneMasti { get; set; }
    [JsonProperty(PropertyName = "seceri")]
    public decimal Seceri { get; set; }
    [JsonProperty(PropertyName = "vlakna")]
    public decimal Vlakna { get; set; }

    [JsonProperty(PropertyName = "cosmosEntityName")]
    public string CosmosEntityName { get; set; }
}
```

Model podataka čini sledećih četiri entiteta u bazi: **recepti**, **koraci_pripreme**, **sastojci** i **namirnice**. Podaci koji se čuvaju o receptima su:

- **id (string)** – identifikator recepta
- **naziv (string)** – naziv recepta
- **opis (string)** – opis recepta
- **datumKreiranja (dateTime)** – datum i vreme kreiranja recepta
- **datumAzuriranja (dateTime)** – datum i vreme ažuriranja recepta

Podaci o koracima pripreme su:

- **id (string)** – identifikator koraka pripreme
- **idRecepta (string)** – identifikator recepta kojem korak pripreme pripada
- **redniBroj (uint)** – redni broj koraka pripreme
- **opis (string)** – opis koraka pripreme
- **savet (string)** – savet prilikom pripreme koraka
- **datumKreiranja (dateTime)** – datum i vreme kreiranja koraka pripreme
- **datumAzuriranja (dateTime)** – datum i vreme ažuriranja koraka pripreme

Podaci o sastojcima su:

- **idRecepta (string)** – identifikator recepta kojem sastojak pripada
- **idNamirnice (string)** – identifikator namirnice koja se koristi kao sastojak
- **kolicina (unit)** – količina namirnice u sastojku
- **jedinicaMere (string)** – jedinica mere količine
- **kolicinaUGramima (unit)** – količina u gramima
- **napomena (string)** – posebna napomena o sastojku
- **datumKreiranja (dateTime)** – datum i vreme kreiranja sastojka
- **datumAzuriranja (dateTime)** – datum i vreme ažuriranja sastojka

Podaci o namirnicama su:

- **`id (string)`** – identifikator namirnice
- **`naziv (string)`** – naziv namirnice
- **`opis (string)`** – opis namirnice
- **`kategorija (string)`** – kategorija namirnice
- **`kalorije (decimal)`** – broj kalorija u 100 grama namirnice
- **`proteini (decimal)`** – količina proteina u 100 grama namirnice
- **`masti (decimal)`** – količina masti u 100 grama namirnice
- **`zasiceneMasti (decimal)`** – količina zasićenih masti u 100 grama namirnice
- **`seceri (decimal)`** – količina šećera u 100 grama namirnice
- **`vlakna (decimal)`** – količina vlakna u 100 grama namirnice

4.2 Postavljanje servisa na platformu

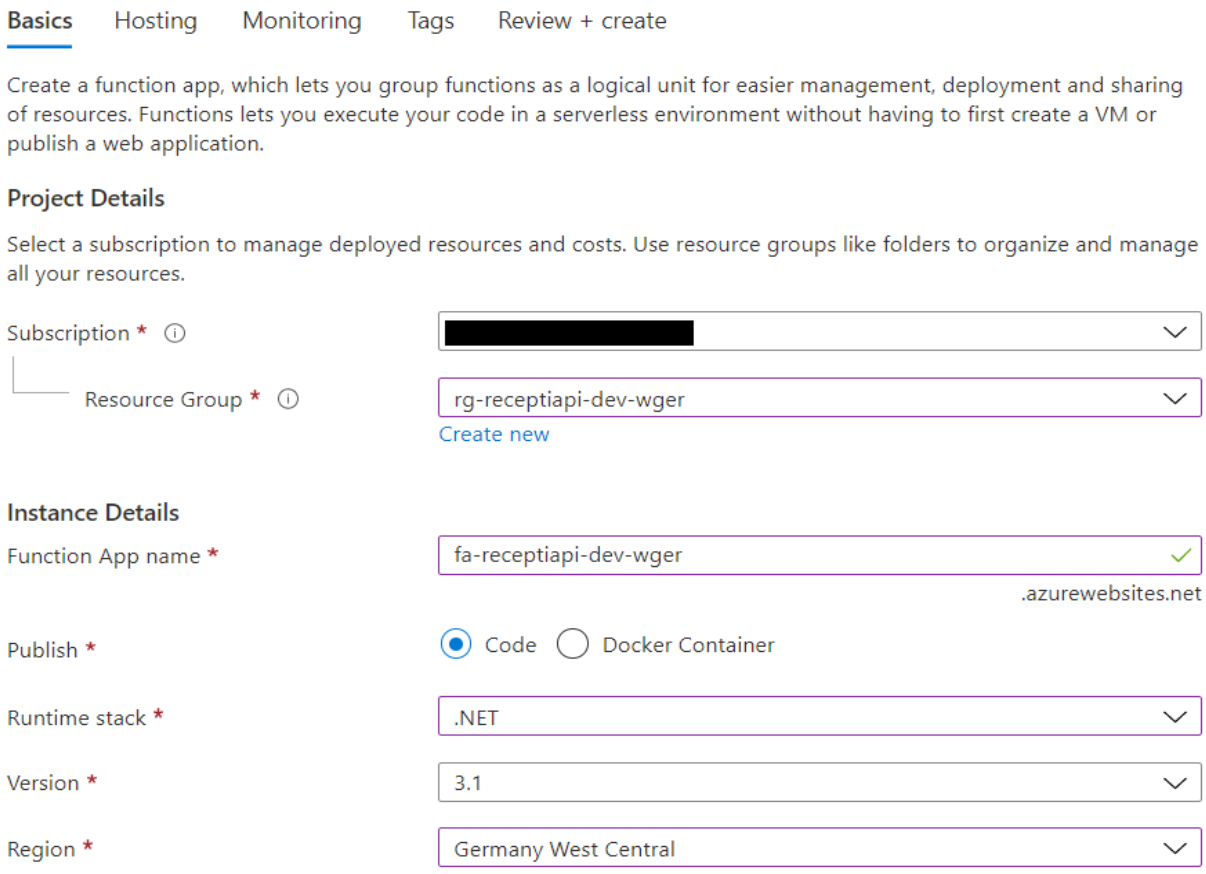
U ovom delu dati su detalji rezervisanja resursa na platformi *Microsoft Azure*, postavljanje servisa "Recepti API" na platformu i testiranje. Za ove potrebe kreiran je besplatni nalog na platformi *Microsoft Azure* koji pruža mogućnost limitirnog korišćenja velikog broja servisa [10]. Kreiranje resursa biće prikazano kroz veb portal platforme, a za postavljanje servisa korišćen je servis *Kudu* [11].

4.2.1 Resursi na platformi

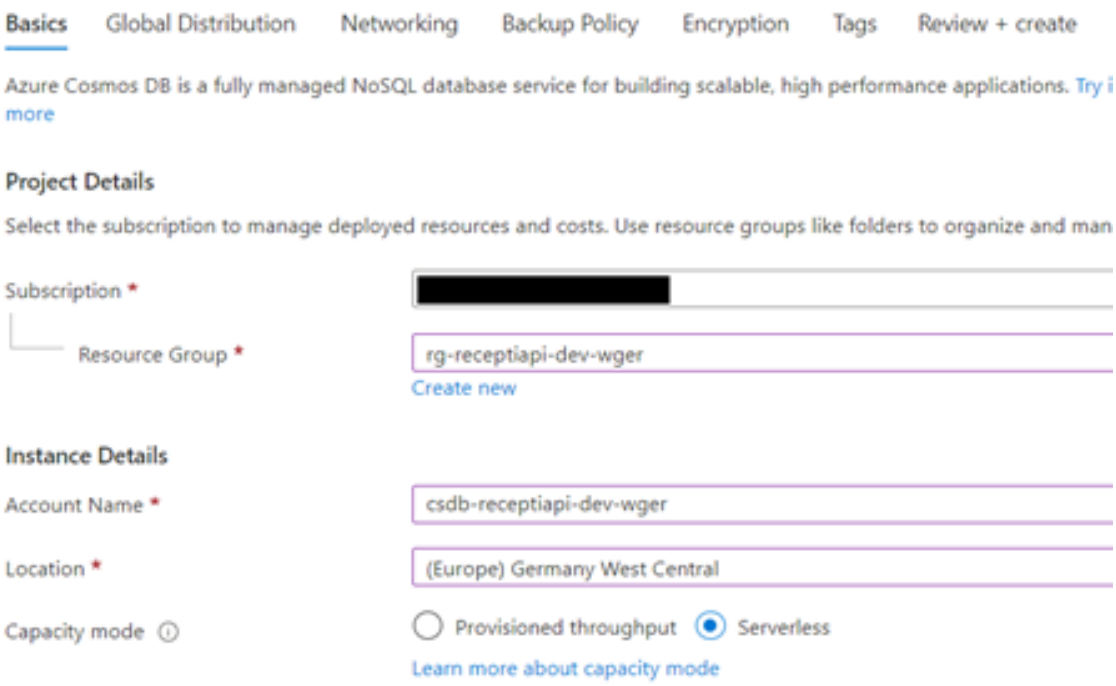
Prilikom imenovanja resursa korišćena je sledeća konvencija: **`tip_resursa-naziv_projekta-okruženje-region`**. Svi resursi servisa jednog okruženja organizovani su u okviru iste resursne grupe zbog lakšeg upravljanja.

Kreiranje projekta funkcija na platformi *Azure Function* koji će se koristiti za hostovanje komponente *ReceptiAPI* prikazano je na Slici 9. Tokom ovog koraka je kreiran i nalog na servisu za skladištenje gde se čuva kod projekta i datoteke dnevnika. Napomena da je prilikom kreiranja automatski postavljena poslednja

verzija radnog okvira *.Net Core* 3.1, ali je nakon toga moguće ažurirati i izabrati prethodne verzije. Za bazu podataka *ReceptiDb* kreiran je nalog na servisu *CosmosDb* sa modelom korišćenja "bez servera", kao što je prikazano na Slici 10.



Slika 9. Kreiranje projekta funkcija na platformi



Slika 10. Kreiranje naloga na servisu CosmosDb za potrebe rezervisanja baze podataka ReceptiDb

4.2.2 Postavljanje servisa i testiranje

Za postavljanje lokalno razvijene verzije servisa korišćen je servis *Kudu*. Ovaj servis omogućava podizanje projekta, pregled datoteka dnevnika, izvršavanje skripti i podešavanje različitih parametara za projekte na platformama *Azure Functions* i *Azure App Service*. Nakon postavljanja i podešavanja parametara okruženja i parametara za povezivanje sa bazom podataka servis je javno dostupan na adresi https://fa-receptiapi-dev-wger.azurewebsites.net.

Primer poziva prikazan je na Primeru koda 13. Kolekcija za alat *Postman* za testiranje svih funkcionalnosti servisa javno je dostupna na repozitorijumu projekta.

Primer koda 13. Primer poziva funkcije za kreiranje namirnica

```
curl --location --request POST 'https://fa-receptiapi-dev-wger.azurewebsites.net/api/v1/namirnice/' \
--header 'x-functions-key: **********' \
--header 'Content-Type: application/json' \
--data-raw '{
   "naziv": "Jabuka",
   "opis": "Domaca crvena jabuka",
   "kategorija": "Voce",
   "kalorije": 0.63,
   "proteini": 0.044,
   "masti": 0.0019,
   "zasiceneMasti": 0,
   "seceri": 0.0959
}'
```

5 Zaključak

U radu su prikazani ključni koncepti računarstva i arhitekture "bez servera", sa posebnim fokusom na model izvršavanja "funkcija kao servis". Opisane su arhitekture i tehnologije koje su uticale na njihov nastanak i način funkcionisanja. Pored toga dat je kratak pregled platformi "funkcija kao servis", njihovih osobina i prednosti i nedostataka. Ovaj način izvršavanja može doneti dosta benefita u vidu smanjenja troškova i operacionih zadataka, kao i automatskog skaliranja bez uticaja programera. Sa druge strane cena koja se mora platiti su vremensko ograničenje zahteva, nepostojanje stanja i potencijalna zavisnost od isporučioca platforme. Iako je arhitektura "bez servera" relativno nov koncept, već sada ona predstavlja jednu od bitnijih alternativa za dizajn arhitekture sistema u oblaku ili nekih delova ovakvih sistema.

U drugom delu rada detaljnije je predstavljena *Azure Functions*, kao platforma otvorenog koda i jedna od tri najkorišćenije platforme ovog tipa. Prikazan je način rada i razvoj funkcija i aplikacija funkcija i njihovo postavljanje na platformu. U praktičnom delu rada razvijen je servis "Recepti API" kao primer jednog dobro organizovanog projekta funkcija. Servis je u potpunosti baziran na arhitekturi "bez servera" i njegov smisao je prikazivanje koncepata i principa modela "funkcija kao servis" i platforme *Azure Functions* koji su opisani u prethodnim poglavljima ovog rada.

Broj korisnika platformi u oblaku svakodnevno raste, a sa njima i broj aplikacija koje se razvijaju primarno za izvršavanje u oblaku. Ovo dovodi do tržišne konkurencije među isporučiocima platformi, a takođe i do zainteresovanosti zajednice otvorenog koda za arhitekturu "bez servera". Mišljenje autora ovog rada je da će sve navedeno doneti mnoga poboljšanja na ovom polju u godinama koje dolaze.

Literatura

[1] "Gartner," 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[2] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015.

[3] M. J. Kavis, Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS), Wiley, 2014.

[4] P. C. K. C. P. C. S. F. V. Ioana Baldini, "Serverless Computing: Current Trends and Open Problems," IBM Research, Bentley University, 2017.

[5] K. Chowhan, Hands-On Serverless Computing, Build, run and orchestrate serverless applications using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions, Packt Publishing, 2018.

[6] Microsoft, "Azure Functions Consumption Plans," 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

[7] S. Rosenbaum, Serverless computing in Azure with .NET, Packt Publishing, 2017.

[8] "Azure Functions Core Tools," 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local.

[9] "Cosmonaut," 2020. [Online]. Available: https://github.com/Elfocrash/Cosmonaut.

[10] "Azure Free Tier," Microsoft, 2021. [Online]. Available: https://azure.microsoft.com/en-us/free/.

[11] "Kudu," 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/app-service/resources-kudu.