Seminarski rad iz SOA: Nuclio

Uroš Stojković

Luka Mladenović

Šta je zapravo Nuclio?

Nuclio je "serverless" framework visokih performansi fokusiran na obradu podataka, U/I operacije i izračunavanju intenzivnih matematičkih operacija integracije.

Dobro je integrisan sa popularnim alatima za nauku o podacima, kao što su Jupyter i Kubeflow;

Podržava mnoštvo izvora podataka i protoka podataka; i podržava izvršenje nad CPU i GPU-ovima

Projekat Nuclio počeo je 2017. Mnogi startap-ovi i preduzeća sada koriste Nuclio u proizvodnji.

Šta znači “serverless”?

Serverless computing se odnosi na način kreiranja i pokretanja aplikacija koji ne zahteva upravljanje serverom.

Ovaj pojam označava model po kome se aplikacija, upakovana u jednu ili više funkcija, otpremi na platformu, a zatim se izvršava, skalira i naplaćuje srazmerno potrebama u tom trenutku.

Sam naziv “serverless” je pomalo nesrećan jer implicira da se serveri ne koriste u izvršavanju aplikacija, što naravno nije tačno.

Termin ima smisla iz ugla korisnika koji postaju oslobođeni obezbeđivanja servera, održavanja, ažuriranja, skaliranja i planiranja kapaciteta. Svi ovi zadaci su obrađeni od strane serverless platforme i kompletno apstrahovani od developera i operations inženjera.

Šta znači “serverless”?

Postoje dve glavne uloge:

- Developer - piše kod i koristi usluge serverless platforme koja mu pruža tačku gledišta iz koje nema servera i iz koje se njegov kod ne izvršava stalno.
- Provajder - obezbeđuje serverless platformu korisnicima

Kao što je već napomenuto, serveri su neophodni za izvršavanje serverless platforme - provajder je taj koji upravlja njima (serverima, virtuelnim mašinama ili kontejnerima). Takođe, izvršavanje platforme će nositi određene troškove za provajdera, čak i kada je platforma besposlena.

Šta znači “serverless”?

Preovladavajuća ideja prilikom dizajniranja web aplikacija bila je da mi, kao programeri, imamo neki određeni nivo kontrole nad HTTP zahtevima koji dolaze do našeg servera.

Naša aplikacija je pokrenuta na datom serveru i mi smo odgovorni za pribavljanje i nagledanje svih resursa koji mogu biti u upotrebi tokom izvršavanja

Ovo bi bila “server-full” arhitektura takoreći

Međutim, ovde postoje određeni problemi, naime:

- Server mora biti pokrenut, čak i kada nema nadoilazećih zahteva
- Problemi vezani za skaliranje, održavanje i bezbednost datog servera

“Serverless” vs. “Serverfull” arhitekture

Ovaj model arhitekture tj. tačnije izvršavanja same aplikacije, odgleda se u upotrebi cloud provajdera, naime AWS, Azure, Google Cloud-a za izvršavanje nekog dela našeg koda, tako što bi se potrebni resursi obezbedili dinamički, u trenutku upotrebe

Naplaćivanje ove usluge bi bilo srazmerno količini resursa koji su upotrebljeni

Kod se izvršava u “stateless” kontejnerima i njihovo izvršavanje mogu “okinuti” različiti događaji:

http zahtevi, događaji vezani za baze podataka, servisi za message queues, alerti kod monitora podataka, upload fajlova, cron job događaji i sl.

Serverless servisi

Serverless racunarska platforma obično nudi jednu ili obe navedene usluge:

- FaaS (Function-As-A-Service) - tipično omogućava računarstvo vođeno događajima. Developeri pišu delove koda koji se okidaju događajima ili HTTP zahtevima.
- BaaS (Backend-As-A-Service) - API-bazirani servisi trećih strana koje nude implementaciju nekog podskupa funkcionalnosti u aplikaciji. Pošto se API nudi kao servis koji se automatski skalira, korisniku se čini kao serverless. Dobar primer ovakvog tipa servisa je Firebase.

Nedostaci serverless platformi cloud provajdera

- Slabe performanse, problem tzv. hladnog starta i izostanak konkurentnog izvršavanja ne odgovaraju potrebama aplikacija koja rade u realnom vremenu
- Prevelika vezanost za izvore događaja i servise specifične za konkretnu platformu - ograničeni smo na specifična skladišta podataka (baze), alate za logiranje i nadgledanje
- Kompleksnost održavanja stanja
- Nemoguć razvoj, dibagiranje, test, kao i deployment u multi-cloud ili hibridnom okruženju

Šta bismo želeli?

- Napraviti serverless arhitekturu fokusiranu od početka na paralelizam, iskorišćenost CPU i memorije, kao i IO operacije.
- Definisati zajednički format događaja nezavisan od tipa događaja (HTTP, Kafka, RabbitMQ, ili bilo koji drugi tip)
- Razdvojiti izvore događaja i podataka od koda funkcija
- Obezbediti veću sigurnost podataka i kontrolu pristupa
- Posvetiti pažnju dibagiranju i logiranju

“Serverless” vs. “Serverfull” arhitekture

Kod koji je potrebno izvršiti se šalje cloud provajderu kako bi se tamo izvršio u vidu funkcije

Otud se “serverless” arhitektura naziva i “Function as a Service” ili FaaS

Naravno, serveri su i dalje potrebni i koriste se prilikom izvršavanja koda, samo što su oni sada apstraktni samom programeru i time mu omogućavaju da ne brine o tim stvarima

Ovaj način projektovanja aplikacije sa sobom donosi par novih paradigmi koje programer mora usvojiti kako bi na najbolji način upotrebio prednosti ovog pristupa

Mikroservisi

Najveća promena se odgleda u tome da prilikom prelaska na “serverless” arhitekturu naša aplikacija mora biti projektovana u vidu “funkcija”, gde smo dosad aplikacije projektovali u vidu monolita, odnosno iako je kod podeljen u fajlove i foldere, kompajluje se zajedno i izvršava se kao jedan proces.

U “serverless” pristupu, moramo preći na mikroservise tj. na organizaciju naše apliakcije kao veći broj manjih, što više nezavisnih celina gde se svaka može pokrenuti u svom kontejneru zasebno.

Stateless funkcije

Funkcije tj. delove koda koje budemo slali na izvršenje, cloud provajder će pokretati u bezbednim, (skoro pa) “stateless” kontejnerima

Ovo znači da neće moći da izvršimo kod na našem aplikacionom serveru koji se izvršava dosta kasnije nakon završavanja nekog eventa ili funkcije koje koriste kontekst od ranije kako bi obradile neki zahtev (nejasno)

Moramo pretpostaviti da će se naše funkcije izvršavati u potpuno novom kontejneru, svaki put kada ih pozovemo

“Cold starts”

Pošto se naše funckije pokreću unutar kontejnera koji će se inicijalizovati na naš zahtev, postoji određeni stepen latencije između ove dve akcije

Ova latencija se naziva “Cold start”

Neki provajderi će održati kontejnere “u životu” neko vreme nakon obrade samog zahteva, kako bi, ako do zahteva dođe u tom periodu, odgovor bio što brži

Povratak na Nuclio

Nuclio se može koristiti kao samostalni Docker kontejner ili se može ugraditi u proizvoljni Kubernetes klaster

Nuclio funkcije se mogu kreirati kroz kod, kroz recimo Jupyter Notebook;

Takođe, Nuclio je ugrađen u MLRun biblioteku koja se primenjuje u data science-u za automatizaciju i nagledanje pipeline-a podataka

Nuclio se ugrađen i u KuberFlow Pipeline framework koji se koristi za kreiranje i deployment portabilnih i skalabilnih ML aplikacija

Još par osobina Nuclio-a

Zahvaljujući svojoj arhitekturi, Nuclio je izuzetno brz: Jedna jedina instanca funkcije u Nuclio može obraditi na stotine hiljada HTTP zahteva ili zapisa podataka u sekundi; To je 10-100 puta veća brzina od većine framework-a.

Nuclio je takođe i obezbeđen; Integrisan je sa Kaniko-om, alatom koji služi za kreiranje Docker image-a iz Dockerfile-ova u Kubernetes klasteru, kako bi obezbedio bezbedan i pouzdan način za kreiranje Docker image-a tokom run-time koji su spremni za deployment

Zašto uopšte baš Nuclio?

Dosad, nijedno od cloud i open-source serverless rešenja nisu uspela da reše neke od gorućih zahteva zajednica, naime:

Procesiranje u realnom vremenu sa minimalnom upotrebom CPU/GPU-a ili I/O sistema dok je paralelizam maksimiziran

Lako integrisanje sa različitim izvorima podataka, trigera, modela obrade i postojećih ML frameworka

Stateful funkcije fokusirane na brzoj obradi velike količine podataka

Prosto debagiranje, testiranje i CI/CD pipeline-ovi

Portabilnost na manje uređaje, laptopove, on-prem ili javne kloud klastere

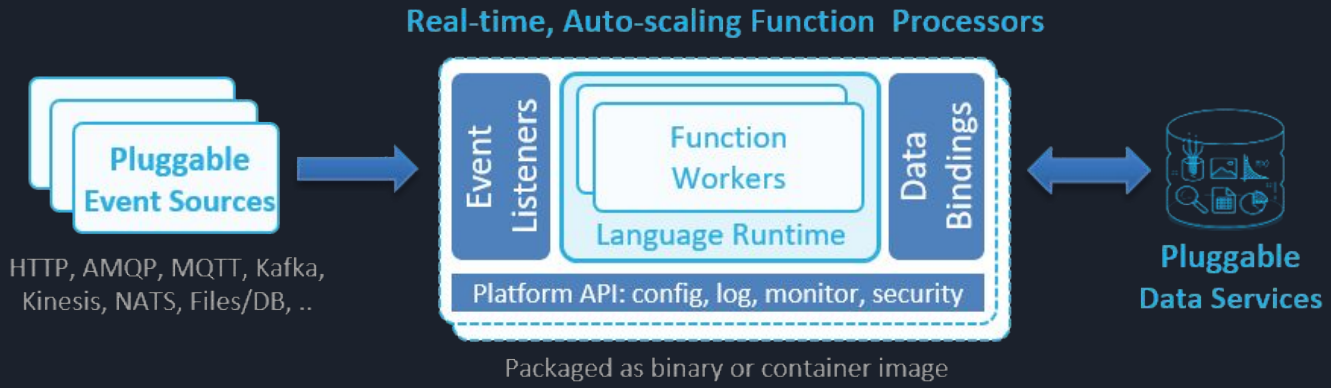
Open-source ali sa enterprise mogućnostima

Zašto uopšte baš Nuclio?

Nuclio je uspeo da ispuni sve ove zahteve

Dizajniran je s namerom da bude proširljiv, open-source framework, koji koristi modularan i slojevit pristup dizajniranja kako bi podržao konstantno dodavanje novih okidača i izvora podataka, sa nadom da će i drugi(m) pomoći u kreiranju novih modula, alata i platformi za Nuclio

Nuclio arhitektura



Nuclio arhitektura-Function processors

Služe da obezbede okruženje za izvršavanje funkcija. Procesor dostavlja funkciji događaje, obezbeđuje kontekst i podatke, sakuplja logove i statistiku i nagleda životni ciklus same funckije

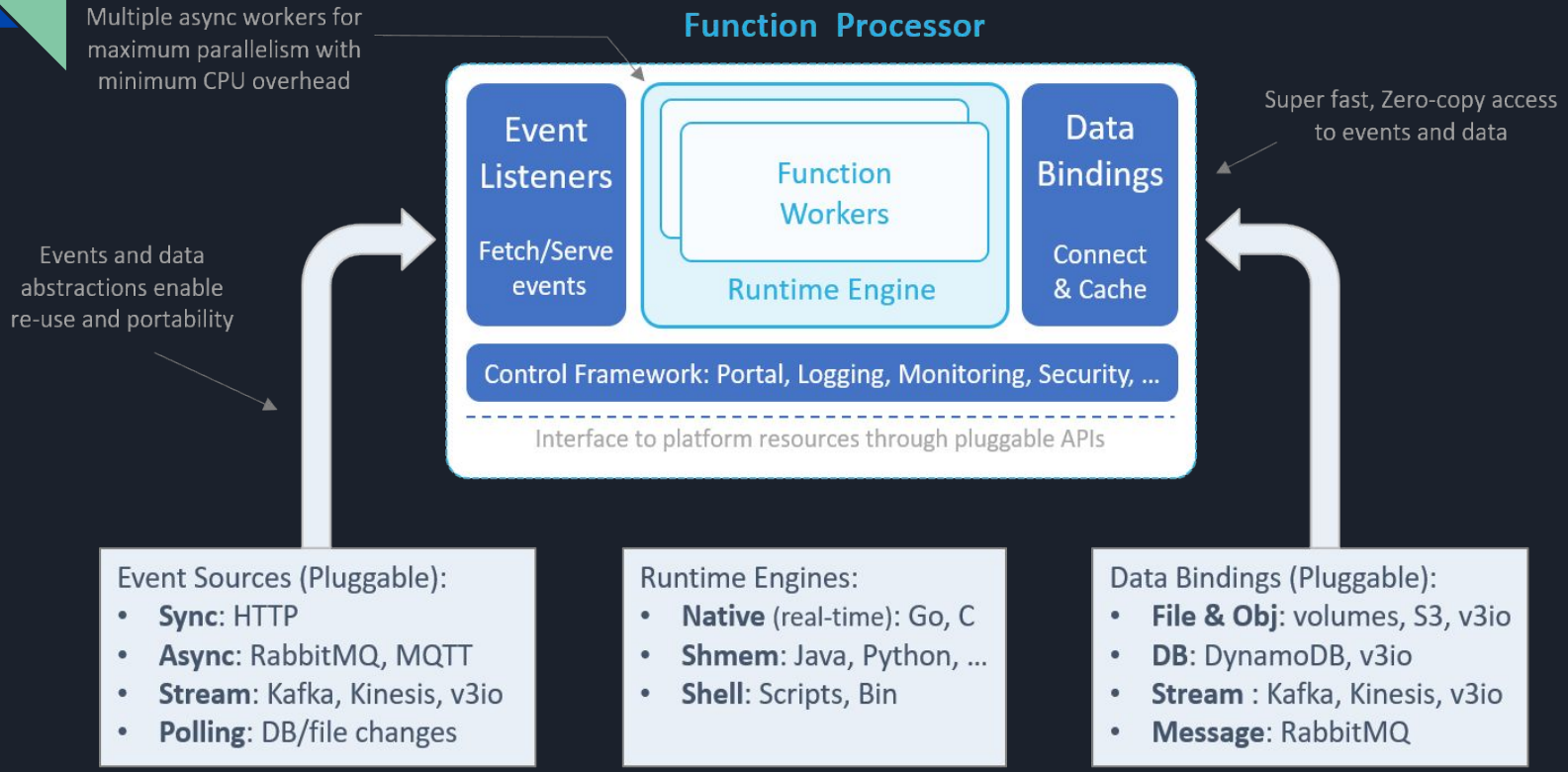
Procesori mogu biti kompajlovani u jedan binarni fajl (Go ili C) ili mogu biti unutar kontejnera zajedno sa svim zavisnostima potrebnim za izvršavanje koda

Kontejneri sa funkcijskim procesorima mogu biti samostalni kontejneri ili mogu biti deo Kubernetes klastera

Svaka funkcija ima svoj procesor

Ako za tim ima potrebe, kontejneri sa procesorima datih funkcija se mogu scale-out-ovati tj. kreiraće se više instanci istog kontejnera kako bi se obradili svi zahtevi

Arhitektura procesora



Arhitektura procesora

Ideja kod Nuclio-vog procesora je maksimizacija performansi same funkcije i obezbeđivanje apstrakcije i portabilnosti kroz široki spektar platformi, izvora samih događaja i servisa vezanih za podatke

Procesor ima četiri glavne komponente:

1 - Event-source listener-e

2- Runtime engine

3- Data bindings

4- Control framework

Event-source listener

Osluškuju sockete i message queues ili periodično fetch-uju evente od nekog eksternog eventa (event source-a) ili izvora podataka

Primljeni eventi imaju zajedničku šemu, koja služi da odvoji logiku same funkcije od implementacije date funkcije <na izvoru samog eventa ili neke specifične strukture> i da je prosledi jednom ili više paralelnih runtime “radnika”

Evente listener-i takođe garantuju exactly-once ili at-least-once izvršavanje event-a i otporni su na otkaze

Runtime engine

Odgovoran je da inicijalizuje funkcijsko okruženje tj. promenljive, kontekst, log, data bindings i sl

Prosleđuje event objekte čvorovima radnicima koji izvršavaju funkcijski kod i vraća odgovor izvoru koji je generisao event

Runtimes mogu imati više paralelnih radnika kako bi obezbedili ne-blokirajuće operacije i maksimalno iskoristili CPU

Nuclio podržava tri tipa procesorskih runtime implementacija:

1- Native, za inline Go ili C rutine

2- SHMEM, za jezike sa deljenom memorijom kao npr Python, Java, Node.js. Procesor komunicira sa SHMEM runtime-mom funkcije kroz zero-copy kanale deljene memorije

3- Shell, za izvršavanje funckija ili binarnih fajlova preko komandne linije, Nakon što primi događaj, procesor pozove na izvršavanje datu funkciju preko komandne linije i mapira funkcijski stdout ili stderr na izlaz same funkcije

Data bindings

Funkcije mogu dobijati dodatne podatke od raznih izvora, kao što su eksterni fajlovi, objekti, baze podataka ili messaging sistemi. Runtime će inicijalizovati data-servis konekciju na osnovu tipa, URL-a, svojstava i credentials koji su navedeni u specifikaciji funkcije i prosleđuje sve podatke funkciji kroz kontekstni objekat

Data bindings olakšavaju development jer ne postoji potreba za integracijom sa raznim SDK-ovima ili potreba za održavanje konekcije. Omogućavaju takođe da funkcije budu portabilne i lako prenosive jer se različiti data servisi iste klase mapiraju ka funkciji preko istih API-ja

Mogu podržati keširanje, micro-batching, smanjuju latenciju kod I/O operacija

Ne rade serijalizaciju i ne blokirajući su

Control framework

Inicijalizuje i kontroliše sve delove procesora, služi kao logger za funkcije i procesor, u odvojenim stream-ovima, nadgleda statistiku samog izvršavanja i služi kao mini portal za remote nadgledanje i upravljanje

Koristi apstraktne interfejse za interakciju sa ostatkom platforme, što omogućava portabilnost na mnogo različitih IoT uređaja, klastera kontejnera i cloud platformi

Fajl “processor.yaml” služi za platform specific konfiguraciju samog procesora

Interfejs samog procesora date funkcije ka ostatku platforme je apstrahovan na način koji omogućava portabilnost

Event sources i mapping

Funckije su event-driven što znači da se “okidaju” na event triggere, data message ili rekorde odnosno podatke koji se prihvataju od event source-a i prosleđuju runtime engine same funkcije

Event sources tj. izvori samih eventa mogu se kategorisati na osnovu njihovog ponašanja i način na koji podaci teku od samog izvora

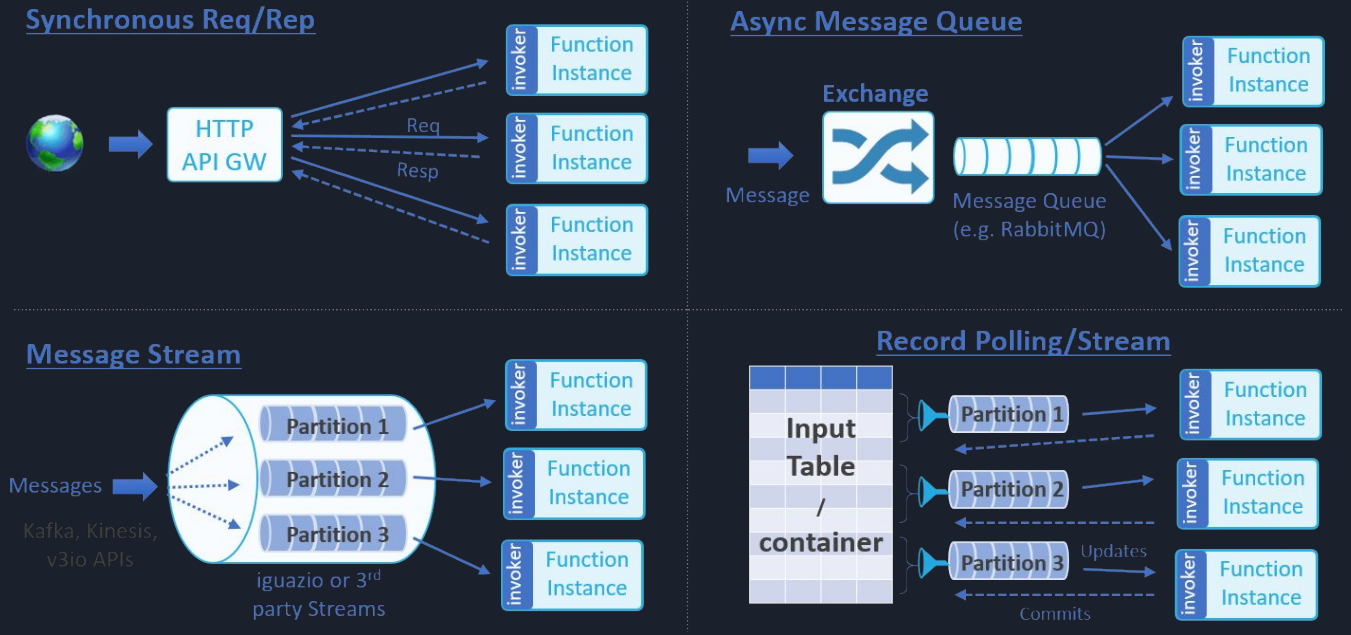
1- Sinhroni Req/Res: klijent pošalje zahtev i čeka na odgovor; recimo HTTP zahtevi ili RPC

2- Asinhroni message-queue zahtev: poruke se “publish”-uju u red i distribuirane su tako do “subscriber”-a. RabbitMQ, emails, cron jobs

3- Message ili Record stream-ovi: uređeni skup poruka ili rekorda koji se sekvencijalno obrađuje. Kafka, AWS Kinesis

4- Record ili Data Polling: skup rekorda ili podataka koji je filterovan na neki način prilikom pribavljanja iz nekog eksternog izvora podataka ili baze podataka. Pribavljanje može biti periodično ili na osnovu nekih triggera

Event sources i mapping



Event-source mapping

Event-sources su mapirane na jedinstvenu verziju funkcije. Npr. API gateway web URL “/” može biti mapiran na production verziju, dok je “/beta” mapiran na test verziju iste funkcije

Event se može odrediti kroz specifikaciju same funkcije ili kroz CRUP API-ja za mapiranje ili kroz CLI komande

Više event izvora mogu biti mapirani na istu funkciju i isti event može okinuti više različitih funkcija

Event load balancing, sharding, dealers

Za izvršenje određenog “posla”, možda je potrebno koristiti više instanci procesora iste funkcije

Recimo distribuirana baza podataka ili Kafka stream može biti podeljen na više instanci funkcije, što zahteva entite zadužene za upravljanje i distribuciju resursa koji će biti korišćeni u svakoj instanci, kao i praćenje toka izvršavanje tih funkcija

Nuclio sadrži tzv. “dealer” entitet koji može dinamički distribuirati N resursa (shard-ova, particija, taskova i sl.) među M procesora i može obraditi greške i skaliranje resursa i procesora u ovom kontekstu

Event objekti

Ovo su objekti koje funkcija koristi

Funckije bivaju pozvane na izvršenje sa dva parametra, kontekst objektom i event objektom

Event objekat opisuje podatke i metapodatke koji su vezani za sam događaj koji je okinuo izvršenje funkcije

Ovi objekti su generalizovani na takav način da odvajaju sam izvor događaja i njegov izgled od izgleda funkcije koja obrađuje taj događaj

Više različitih izvora eventa može okinuti istu funkciju

Po potrebi, funkcije su u stanju da prime jedan event objekat ili niz event objekata

Event objekti

Event objektima se pristupa preko interfejsa tj. metoda samog interfejsa

Event objekti se mogu serijalizoati u JSON objekte što može prouzrokovati dodatni overhead

Neke od tipičnih polja event objekta su EventID, Body, Content-Type, Headers, Fields i AsJson

Za svaku od klasa sami event-a mogu postojati specifčna polja vezane za konkretnu klasu

Kako napraviti funkciju?

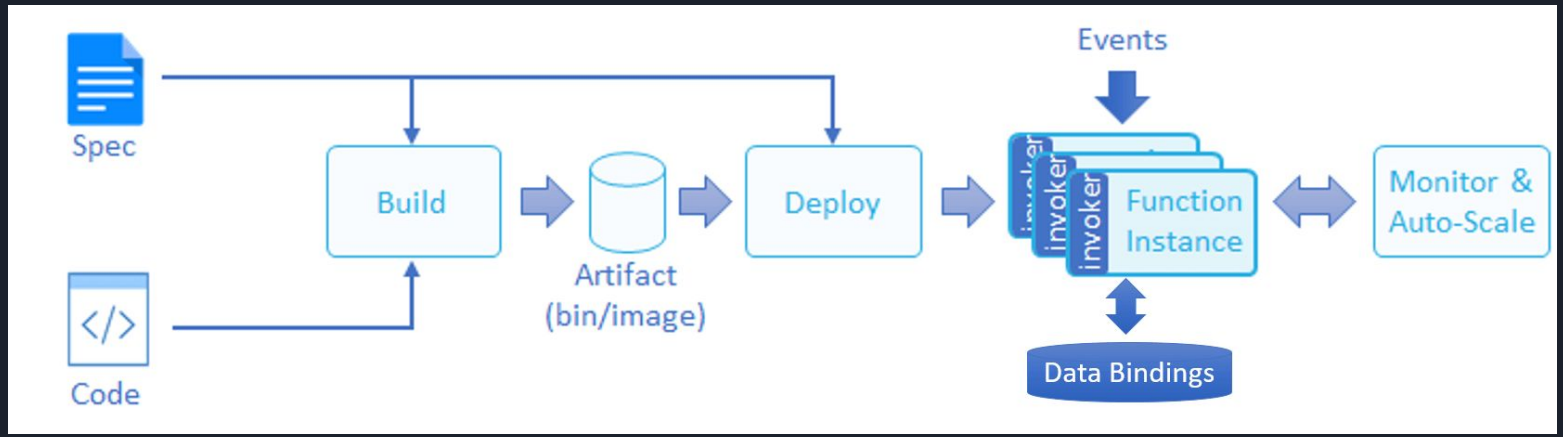
Funkcija se može implementirati u jednom od podržanih jezika

Zatim, potrebno je build-ovati tj. kompajlirati datu funkciju tj. parče koda kao binarni fajl, package ili container image, i dobijenu funkciju deploy-ovati na željeni klaster

Tako deploy-ovanoj funkciji se sada mogu poslati eventi i podaci za dalju obradu

Na sledećem slajdu je prikazan proces kreiranja i deploy-ovanja funkcije

Kako napraviti funckiju?



Kako napraviti funckiju?

Svaka funkcija tj. svaka vezija funckije ima poseban fajl pod nazivom “spec”

On definiše različita svojstva funkcije, kao sam kod funkcije, data bindings, environment resurse i event source-ove

Ovaj fajl može biti napisan u YAML ili JSON formatu i može se editovati kroz CLI opcije

Builder će iskoristiti ovaj fajl u fazi kompajliranja same funkcije, a kontroler će se pozvati na ovaj fajl kako bi identifikovao operacione zahteve koji su potrebni samoj funkciji

Pokretanje i build funkcije

Korisnik je u mogućnosti da izabere da li će se ceo proces koji je pomenut na prethodna par slajda izvršiti odjednom ili u koracima

Komandom “build” funckija će biti kompajlirana, kako bi se kasnije moga iskoristitit u raznim deploymentima

Komanda “run” može prihvatiti event source i može odmah deploy-ovati postojeću kompajliranu funkciju ili može build-ovati i deploy-ovati novu funkciju

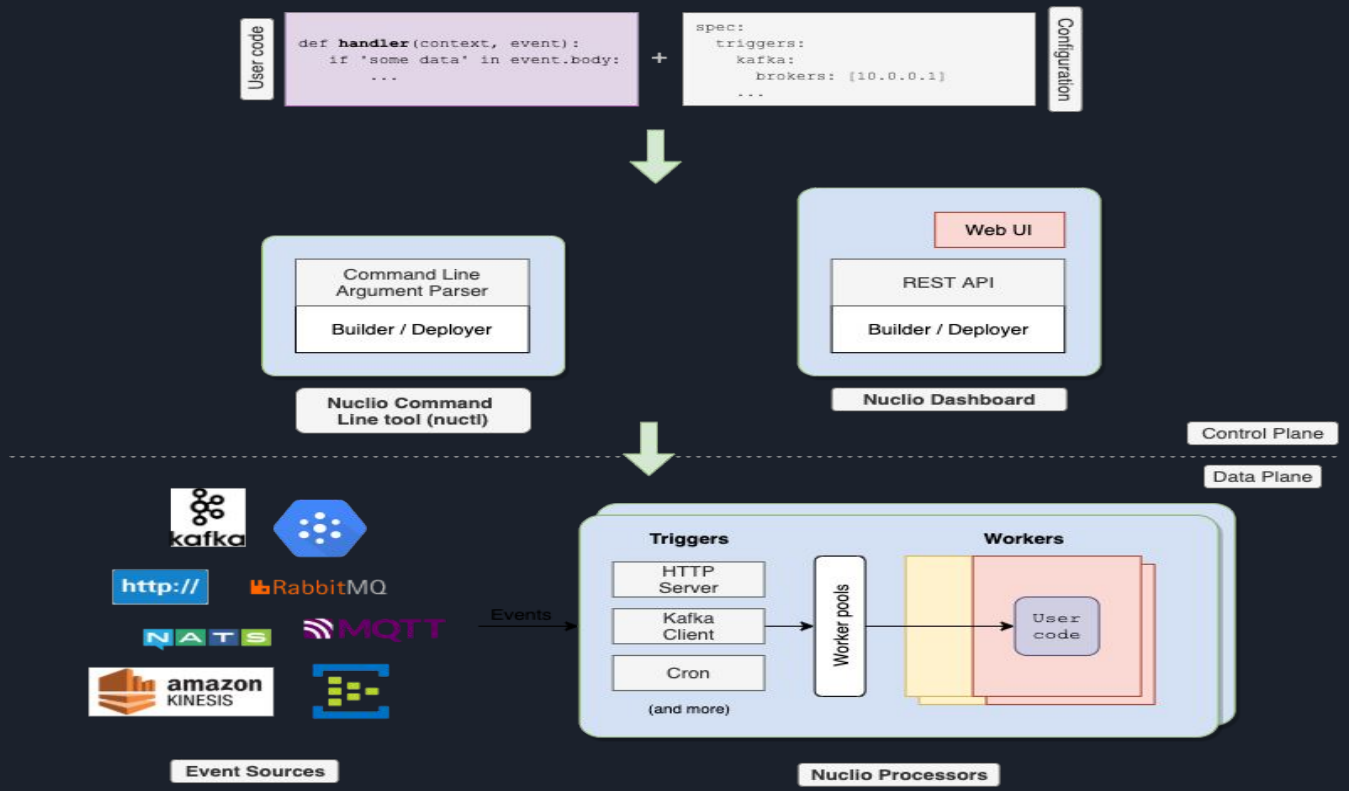
Da rezimiramo...

Nuclio pokušava da apstrahuje što je više moguće podatak vezane za neki event npr. desio se upis u Kafku, poslat je HTTP zahtev, tajmer je istekao i sl. i ove informacije prosledi do konkretne funkcije kako bi ih ona na neki način obradila

Od korisnika je zahtevano da definiše koji tačno event okida koju funkciju, što se može uraditi kroz CLI, REST API ili web aplikaciju

Ovo je prikazano na sledećem slajdu…

Da rezimiramo...



Da rezimiramo...

Nuclio će uzeti ove informacije tj. handler i configuration i proslediće ih function builder-u koji će kreirati container image ove funkcije

Taj image će sadržati kod handlera kao i parče koda koje će izvršiti ovaj handler svaki put kada se naznačeni event dogodi

Builder će zatim “publish”-ovati ovaj image u container registry

Kada je publish-ovan, ovaj image se može deploy-ovati, gde će se u zavisnosti od alata koji se koristi se upravljanje kontejnerima kreirati specifična konfiguracija same funkcije

Da rezimiramo...

Kada je ovaj kontejner sa slikom funkcije ubačen u neki klaster kontejnera, ulazna tačka tj. tačka komunikacije ovakvog kontejnera sa ostatkom sveta je procesor funkcije koji je zadužen za čitanje konfiguracije funkcije, osluškivanje za evente, čitanje eventa kada se oni dogode i pozivanje handlera za dati event

Pored ovog, procesor je zadužen i za još dosta toga, kao što su logovanje, obrade metrika i statistike, otpakivane i zapakivanje req/res, handle-ovanje grešaka i sl

Hvala vam na pažnji!