Mikroservisi part 2

1. REST services

Fokus REST web service-a je na resursima, i kako omogućiti pristup tim resursima. Resurs moze biti predstavljen kao objekat u memoriji, file na disku, podaci iz aplikacije, baze podataka itd. Prilikom dizajniranja sistema prvo je potrebno da identifikujemo resurse, i da ustanovimo kako su međusobno povezani. Ovaj postupaka je slican modelovanju baze podataka. Kada smo identifikovali resurse, sledeći korak je da pronađemo način kako da te resurse reprezentujemo u našem sistemu. Za te potrebe mozemo koristiti bilo koji format za reprezentaciju resursa (JSON, XML npr.). Da bi dobili sadržaj sa udaljene lokacije (od nekog server-a), client mora napraviti HTTP zahteh, i poslati ga web service-u. Nakon svakog HTTP zahteva, sledi i HTTP odgovor od server-a ka client-u tj. onome ko je zahtev poslao. Client move biti korisnik, ili može biti neka druga aplikacija (npr. drugi web service.).

Svaki HTTP zahtev se sastoji od nekoliko elemenata:

- **<VERB>** GET, PUT, POST, DELETE, OPTIONS, itd. odnosno koju operaciju želmo da uradimo.
- **<URL>** Putanja do resursa nad kojim ce operacija biti izvedena.

Kada client dobije odgovor nazad, dobice sadržaj (ako ga ima), ali i **status code**. Ovaj code nam govori da li je prethodno zahtevana operacija izvršena uspesno, ili ne. Status code je reprezentovan celim brojem I to:

- Success 2xx sve je prošlo ok
- Redirection 3xx desio se redirect na neki drugi servis
- Error 4xx, 5xx javila se greska

Svaki servis mora imati jedinstvenu adresu (URL) na koju šaljemo HTTP zahtev na primer:

- http://MyService/Persons/1

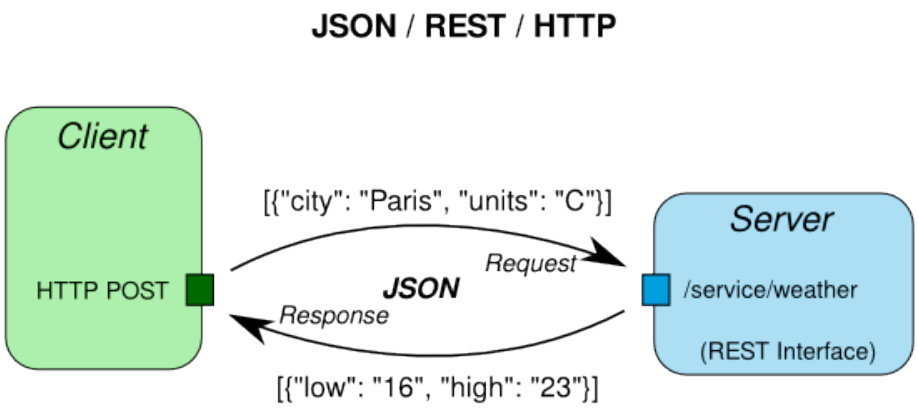
Ako zelimo da izvedemo nekakav upit nad web service-om, to možemo da uradimo koristeći **HTTP VERB** i putanju našeg web service-a ili dodavajući **?** simbol na kraj putanje (ova opcija više simbolizuje RPC nego REST zahtev, ali se može koristiti).

Nakon specijanog simbola, slede parovi u obliku **key=value** spojeni **&** simbolom ako tih parametara ima vise od jednog na primer:

- http://MyService/Persons/1?format=json&encoding=UTF8

Servisi koje pozivamo preko internet imaju već definisanu strukturu (tačnu putanju metod kojim se pozivaju, podatke koje očekuju, kako se pretražuju). AKo mi implementiramo web service-e, onda mi namećemo ova pravila.

Slika 1, predstavlja primer client-server komunikacije.



Slika 1. Client-server komunikacija

2. JSON: JavaScript Object Notation

JSON je format za laksu razmenu podataka, kao i XML nezavistan je od programskog jezika i tehnologije koja se koristi. Podaci se zapisuju kao parovi **ključ:vrednost**.

Ključ se navodi kao tekst pod duplim navodnicima nakon cega sledi vrednost na primer:

- “firstName":"John"

JSON vrednosti mogu biti neke od unapred definisanih:

- A number (integer or floating point)
- A string (in double quotes)
- A Boolean (true or false)
- An array (in square brackets)
- An object (in curly braces)
- Null

JSON **object** se zapisuje u parovima **ključ:vrednost** koji se nalaze unutar vitičastih zagrada:

- {"firstName":"John", "lastName":"Doe"}

JSON Array sadrži ključ, nakon čega sledi niz elemenata u uglasim zagradama:

```
"employees":[
        {"firstName":"John", "lastName":"Doe"},
        {"firstName":"Anna", “lastName”:”Smith"},
        {"firstName":"Peter","lastName":"Jones"}
]
```

JSON može da kombinuje razne tipove podataka unutar jednog niza, o tome treba voditi računa kada koristimo strogo tipizirane jezike da ne bi doslo do problema prilikom konverzije. Mešanje tipova treba izbegavati.

3. REST services u golang-u

Golang u svojoj standardnoj biblioteci ima već ugrađenu podršku za implementaciju mrežnih aplikacija, samim tim i web service-a. Standardna biblioteka je sasvim dovoljna za implementaciju, međutim, da bi olakšali posao možemo koristiti neki od dostupnih bibliteka Gin, Gorilla, i tako dalje. U primerima biće korišćena biblioteka Gorilla.

A. Instalacija biblioteka

Golang ima nekoliko alata za instalaciju paketa i zavisnosti, ovde će biti korišćen alat koji se zove **mod** i dostpuan je u standardnoj bibliotekci od verzije 1.11. Go mod je jednostavan za korišćenje i ima nekoliko komandi:

- **go mod init**, koristi se da inicijalizuje prazan **go.mod** file u direktorijumu projekta. Ovaj file sadrži spisak svih potrebnih biblioteka vačem go projektu.
- **go get**, dobaljna novi biblioteke ili instalira nove.
- **go mod tidy**, briše biblioteke koje se ne koriste.

Slika 2 predstavlja primer go. mod file-a

```
module github.com/milossimic/rest

go 1.15

require (
  github.com/gorilla/mux v1.8.0
  github.com/opentracing/opentracing-go v1.2.0
  github.com/pkg/errors v0.9.1 // indirect
  github.com/uber/jaeger-client-go v2.25.0+incompatible
  github.com/uber/jaeger-lib v2.4.1+incompatible
  go.uber.org/atomic v1.7.0 // indirect
  gorm.io/driver/postgres v1.0.8
  gorm.io/gorm v1.21.6
)
```

Slika 2. primer go.mod file-a

A. Paket context

Paket **context**, se nalazi unutar golang-ove standardne biblioteke i on sadrži samo jedan tip **Context**. Ovaj tip, nam pruža dosta stvaru koje su jako bitne za mrežnu, procesnu komunikaciju unutar aplikacija.

Tip **Context**, se dosta koristi unutar golang aplikacija, zato sto nam pruža jedinstvenu mogućnost da prenosimo podatke, signale za prekid izvršavanja svih povezanih učesnika u komunikaciji. Kada zahtev stigne na nekakv web service i kada krenemo izvršavanje kod-a koji će izvršavati raličite operacije na primer logging, upiti ka bazi, nekakve util funkcionalnosti itd, svakom ovom pozivu možemo da posaljemo i podatak tipa Context i na taj način stvaramo **graf poziva**.

Ovo je bitno, zato sto ako nekakav zahtev traje predugo, ili client prosto odustane od zahteva ili nešto slično tome, golang nam pruža mogućnosti da prekinemo zahtev kao i upite nad bazom i sve ostale zavisne pozive koji su se desili (pod uslovom da je context iskorišćen). Ako na primer znamo da web service treba da odgovori u roku od 30s, a on to ne uradi…context može da emituje event kojim se svi ostali zavisni pozivi prekidaju (tj. bivaju obavešteni o prekidu izvršavanja) i korisniku možemo da vratimo odgovor da rezultata nema ili da server je zauzet ili nešto treće.

B. Graceful shutdown

Prethodno opisani paket, možemo da koristimo implementiramo **graceful shutdown** mehanizam našeg web service-a. Kada korisnik ili operativni sistem prekinu izvršavanje web service-a, mi ne moramo istog momenta da ugasimo naš web server, zato što možta postoje akcije koje se nisu završile.

Dobra je praksa, da se web service-u da neko dodatno vreme u kom on neće prihvatati nove zahteve, ali će ostali elementi moći da urade svoj posao. Na primer, završe svi odgovori korisnicima, završi interkacija sa bazom, sačuvaju logovi itd. Za ovaj mehanizam možemo da iskoristimo prethodni paket….kada dobijemo sistemski poziv, mi možemo da zaustavimo web service da više ne prihvata zahteve, ali ostalim akcijama damo proizviljno vreme da urade sve što treba pre nego što se program načisto zaustavi.

4. Tracing

U aplikacijam koje su distribuirane po svojoj prirodi, vrlo je bitno da znamo šta se desilo tokom svakom korisničkog zahteva. Ako dođe do nekakvog problema, treba da vidimo zašto je do njega došlo i koji je skup poziva koji su to proizveli.

U tradicionalnim aplikacijama, ovo je izuzetno lakše izvodljivo zato što je stanje aplikacije na jednom mestu i direkno iz skupljenih logova možemo da vidimo kada se desio problem i zašto, dok je kod distribuiranih aplikacija ovaj posao znatno teži zato što nije tako jednostavno znati koji poziv se desio pre ili posle nekog drugog.

Da bu rešili ovaj problem, kako da nadgledaju distribuiranu infrastrukturu ljudi iz Googla su došli do metode koja se zove **tracing**.

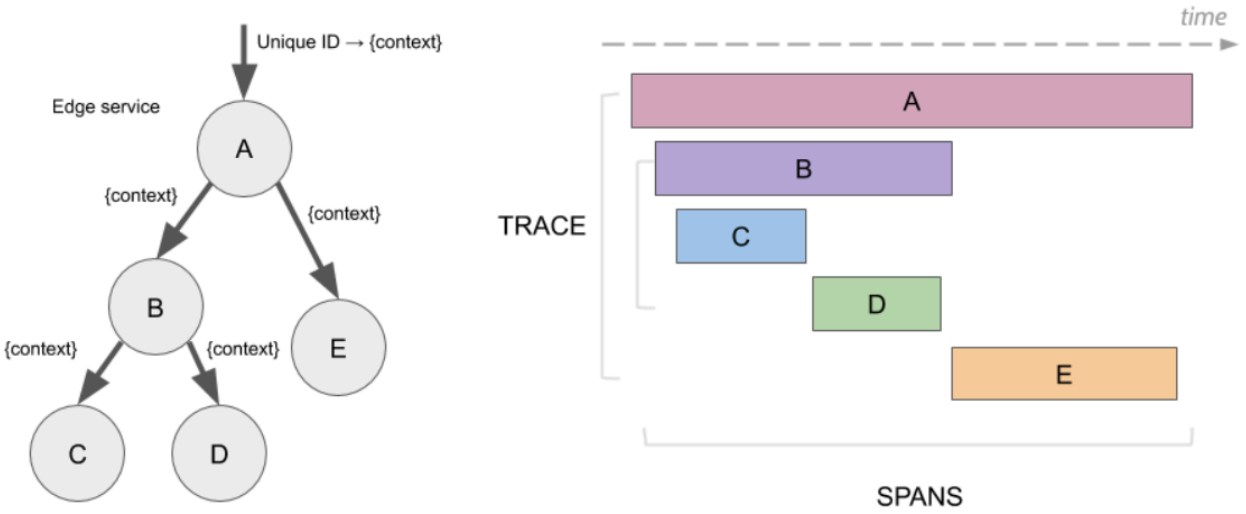
Tracing obuhvata skup podataka koji se šalju svim učenicima u komunikaciji da bi se ustanovilo ko je pozvan pre ili posle koga, odnosno da bi dobili kompletan graf poziva zajedno sa logovima i svim drugim potrebnim elementima i to na jednom mestu.

Ideja iza tracing-a, je vrlo jednostavna. Obeležiti početak i kraj izvršavanja svakog parčeta kod-a u svakom web service-u. Na taj način znamo ko se izvršavao pre koga i možemo da dobijemo jedinstven graf izvršavanja. Svako izvršavanje kod-a se naziva **span**.

Unutar svakog span-a, možemo dodati logove, možemo dobiti vreme izvršavanja svakog parčeta kod-a tako da možemo videti i gde su uska grla našeg kod-a.

Za potrebe ovog predmeta koristiće se opensource alat koji se zove **Jaeger**, i konkretno **all in one docker image**, koji će nam obezbediti i alat za skupljanje informacija i alat za vizuelizaciju toka poziva.

Slika 3 prikazuje primer izvršavanja koda i graf poziva.



Slika 4. Primer izvršavanja sa grafom poziva

Paket context, koji je opisan u prethodnom poglavlju, može se iskoristiti za prenos informacija o grafu poziva unutar metoda/funkcija naše aplikacije.

Ako prethodno nisu definisane informacije o grafu poziva, možemo kreirati nov graf i ubaciti potrebne informacije.

5. Dodatni materijali

1) https://logz.io/blog/go-instrumentation-distributed-tracing-jaeger/
2) https://github.com/albertteoh/jaeger-go-examplehttps://github.com/yurishkuro/opentracing-tutorial
3) https://github.com/alex-leonhardt/go-trace-example
4) https://dwahyudi.github.io/2021/03/21/golang-and-jaeger.html#capturing-errors
5) https://programmer.help/blogs/using-opentracing-and-jaeger-to-realize-link-tracing-of-golang.html
6) https://studygolang.com/articles/25337
7) https://lebum.medium.com/understanding-and-usage-of-context-in-golang-6a460f9e8d2f
8) https://medium.com/@pinkudebnath/graceful-shutdown-of-golang-servers-using-context-and-os-signals-cc1fa2c55e97#:~:text=Graceful%20shutdown%20of%20Golang%20servers%20using%20Context%20and%20OS%20signals,-Pinku%20Deb%20Nath&text=We%20keep%20listening%20for%20OS,whenever%20the%20context%20is%20done.
9) https://kelvinsp.medium.com/building-and-testing-a-rest-api-in-golang-using-gorilla-mux-and-mysql-1f0518818ff6
10) https://gowebexamples.com/routes-using-gorilla-mux/