Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković
Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

(5) Mikroservisna arhitektura

Mikroservisna arhitektura predstavlja pristup dizajnu softvera u kojem se aplikacija razvija kao skup manjih, neovisnih servisa koji međusobno komuniciraju putem mreže. Svaki servis djeluje kao zasebna cjelina zadužena za jasno definiranu funkcionalnost, a komunikacija se najčešće odvija korištenjem standardiziranih protokola poput HTTP-a. Za razliku od tradicionalne monolitne arhitekture, gdje su sve komponente objedinjene u jedinstven sustav, mikroservisna arhitektura razdvaja ključne elemente poput poslovne logike, baza podataka, autentifikacije i drugih funkcionalnosti u samostalne servise.

Takav način izgradnje sustava donosi niz prednosti: omogućuje jednostavnije skaliranje, povećava otpornost na greške, olakšava rad većim razvojnim timovima te ubrzava razvoj, testiranje i implementaciju novih funkcionalnosti. Zbog svoje prilagodljivosti i održivosti, mikroservisna arhitektura predstavlja snažan temelj za dugoročno uspješan razvoj softverskih rješenja, ali istovremeno zahtijeva pažljivo planiranje i upravljanje kako bi se reducirali izazovi povezani s kompleksnošću raspodijeljenih sustava.

🆙 **Posljednje ažurirano: 14.12.2025.**

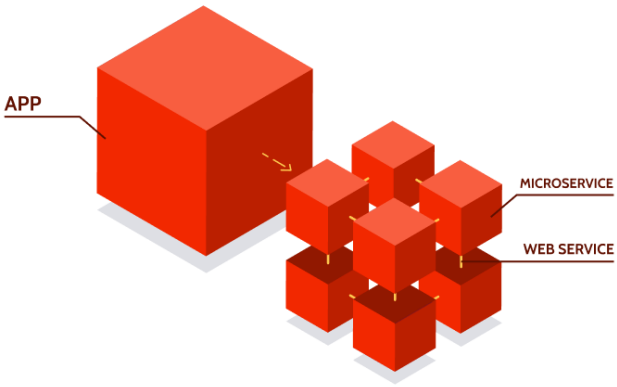
Sadržaj

1. Što je mikroservisna arhitektura?

U softverskom inženjerstvu, **mikroservisna arhitektura** (*eng. microservice architecture*) predstavlja arhitekturalni stil u kojem se aplikacija sastoji od više manjih granularnih servisa, koji komuniciraju putem određenih mrežnih ili RPC protokola. Arhitektura bazirana na mikroservisima omogućava razvojnim timovima da razvijaju i održavaju servise neovisno jedan o drugome, čime se smanjuje interna složenost aplikacije i ovisnost između različitih komponenti, a time i povećava skalabilnost, modularnost i složenost upravljanja razvojnim procesom.

Ne postoji jedinstvena definicija mikroservisne arhitekture, međutim s vremenom je došlo do uspostavljanja konvencija i dobrih praksi koje se primjenjuju u većini slučajeva prilikom njihova razvoja, testiranja i održavanja. Prema tome, možemo definirati nekoliko **ključnih karakteristika mikroservisne arhitekture**:

- U mikroservisnoj arhitekturi, servisi se obično implementiraju kao **odvojeni procesi** koji međusobno komuniciraju putem mreže ili RPC protokola, za razliku od klasičnih monolitnih aplikacija, gdje su sve (ili većina) komponente objedinjene unutar jednog procesa.
- Servisi su osmišljeni tako da se organiziraju oko **poslovnih funkcionalnosti** ili **domenskih entiteta**. Na primjer, možemo imati zasebne servise za upravljanje korisnicima aplikacije, proizvode (*inventory management*) ili narudžbe (*order processing*), pri čemu svaki servis pokriva određeni aspekt poslovanja. Ipak, ne treba pretjerivati i svaki mali dio aplikacije pretvarati u zaseban mikroservis - na taj način bismo brzo bankrotirali zbog prevelikih troškova upravljanja infrastrukturom.
- Glavna ideja mikroservisa je omogućiti njihovu **neovisnu implementaciju i razvoj**. To znači da svaki servis može koristiti različite tehnologije, programske jezike ili baze podataka, ovisno o tome što najbolje odgovara njegovim specifičnim potrebama.
- **Mikroservisi su obično kompaktni**, kako po broju linija kôda, tako i po resursima koje koriste. Razvijaju se i **autonomno isporučuju kroz automatizirane procese**, poput sustava za kontinuiranu integraciju i isporuku (CI/CD), što omogućava bržu i fleksibilniju iteraciju.



> Ilustracija podjele sustava na distribuiranu mikroservisnu arhitekturu

Kao i svaki arhitekturalni stil, mikroservisna arhitektura ima svoje prednosti i nedostatke, samim tim **nije uvijek najbolje rješenje za svaki problem**. Razvoj aplikacije oko mikroservisa često zahtijeva dodatne **inicijalne troškove** i napore u postavljanju infrastrukture, automatizaciji te upravljanju servisima (ali i ljudskim resursima koji stoje iza razvoja).

Monolitna arhitektura, kao klasična alternativa mikroservisnom pristupu, predstavlja način razvoja aplikacije kao jedinstvene, povezane cjeline, obično objedinjene u jednom procesu ili aplikaciji. Ovaj pristup nudi brojne prednosti, uključujući jednostavnost razvoja, održavanja i testiranja. Ipak, kako aplikacija postaje sve složenija zbog povećanja funkcionalnosti i broja korisnika, mogu se javiti izazovi povezani sa

skalabilnošću i prilagodljivošću.

1.1 Monolitna arhitektura

Monolitna arhitektura (eng. *monolithic architecture*) je stil arhitekture u kojem je cijela aplikacija dizajnirana kao "jedinstvena" povezana cjelina. To znači da su svi moduli i komponente aplikacije, poput korisničkog sučelja, poslovne logike, pristupa podacima, postojani u unutar jedne aplikacije. Čista monolitna aplikacija se obično implementira kao jedan veliki "programski paket" ili proces koji se izvozi i pokreće samostalno.

Softverska rješenja koja ste do sad razvijali na kolegijima Programsko inženjerstvo i Web aplikacije, mogla bi se opisati kao monolitne aplikacije, iako one to nisu u pravom smislu definicije monolitnosti. Naime, monolitna arhitektura je često povezana s klasičnim *desktop* aplikacijama, gdje se cijela aplikacija izvršava na korisnikovom računalu, bez potrebe za dodatnim komponentama ili servisima - cijela poslovna logika, pristup podacima i korisničko sučelje su objedinjeni unutar jedne aplikacije - prisjetite se npr. *Java Swing* ili *WPF* aplikacija.

Kako smo na **Programskom inženjerstvu** aplikaciju razvijali u okviru jednog razvojnog okvira (Vue.js), koristeći jedan programski jezik (JavaScript) te koristili Firebase kao servis za autentifikaciju i bazu podataka na način da smo ga integrirali direktno u aplikaciju, možemo argumentirati da smo razvijali aplikaciju u monolitnoj arhitekturi. Međutim, **Firebase je PaaS** (*Platform-as-a-Service*) **usluga**, odnosno platforma u oblaku koja omogućava korištenje udaljenih poslužitelja i nudi razne funkcionalnosti kroz skup Google-ovih mikroservisa koji rade na GCP (*eng. Google Cloud Platform*). Glavna prednost Firebasea je što eliminira potrebu za brigom o infrastrukturi, upravljanju bazama podataka, skaliranju aplikacije i sl. - sve to rješava sam GCP/Firebase. Stoga se može reći da ste, na određeni način, svoju aplikaciju razvijali u okviru mikroservisne arhitekture, ali na višem nivou apstrakcije, gdje je sama platforma (Firebase) bila zadužena za upravljanje mikroservisima u oblaku i skaliranju vaše aplikacije.

Što se tiče **Web aplikacija**, kolegij obuhvaća razvoj klijentske i **poslužiteljske strane aplikacije**. Poslužiteljska strana aplikacije razvijana je prema monolitnoj arhitekturi budući da je sadržavala sve komponente potrebne za uspješan rad aplikacije (poslovnu logiku, pristup podacima, autentifikaciju korisnika i sl.) unutar jedne aplikacije, bez razdvajanja na manje, samostalne servise. Ipak, u praksi se često koristi vanjski servis za pohranu podataka (npr. baza podataka na nekom udaljenom poslužitelju), što može donekle narušiti čistu definiciju monolitne arhitekture.

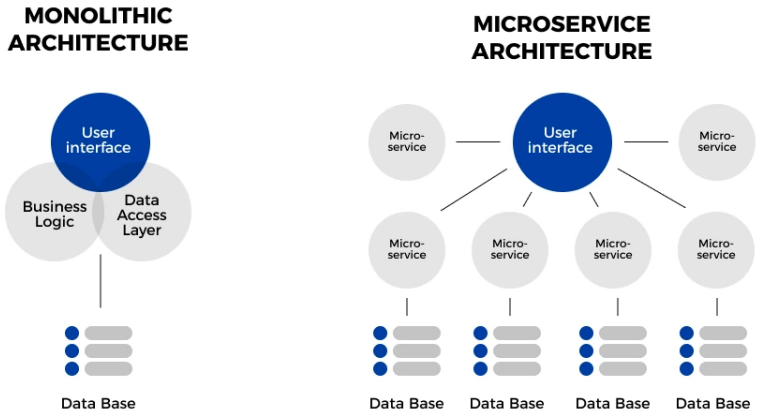
Izazovi povezani s ovakvim pristupom već su prethodno spomenuti: skalabilnost, održavanje, testiranje, razvoj itd.

Neki od čestih problema s kojima se susreću razvojni timovi prilikom razvoja monolitnih i semi-monolitnih aplikacija uključuju:

- *Što ako broj korisnika aplikacije naglo poraste, a postojeća infrastruktura ne može podnijeti opterećenje?*
- *Kako se učinkovito nositi s velikom količinom podataka u bazi?*
- *Kako brzo i sigurno isporučiti nove verzije aplikacije korisnicima bez prekida u radu?*
- *Kako testirati pojedine dijelove aplikacije neovisno jedan o drugome, bez narušavanja korisničkog iskustva?*
- *Što ako mi "padne" cijeli poslužitelj zbog greške u poslovnoj logici jednog dijela aplikacije - korisnici se više ne mogu niti prijaviti?..*

- *Kako organizirati veliki razvojni tim s različitim kompetencijama da učinkovito surađuje na razvoju jedne velike aplikacije?*
- *Razvojni tim nam je heterogen - članovi tima imaju različite preferencije u pogledu programskih jezika, okvira i alata. Kako možemo omogućiti svakom članu tima da koristi tehnologije koje najbolje odgovaraju njegovim vještinama i potrebama, a istovremeno osigurati da svi dijelovi aplikacije rade zajedno bez problema?*

i tako dalje...



> Monolitna vs. mikroservisna arhitektura razvoja aplikacija: *high-level* prikaz aplikacijskih komponenti

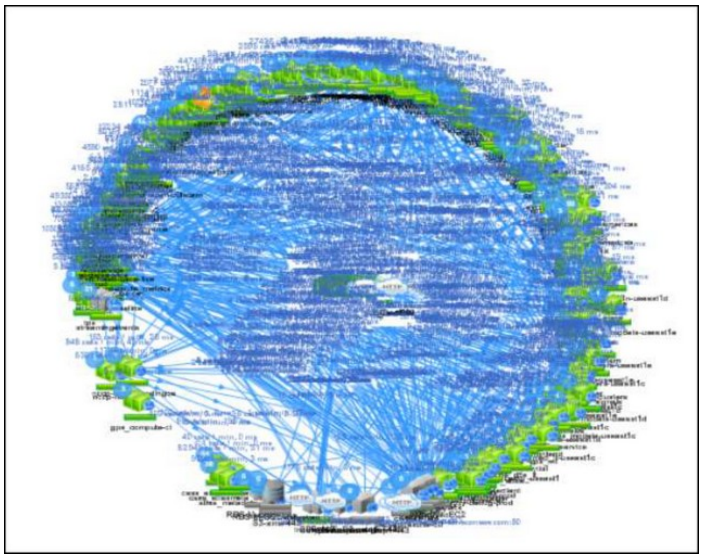
1.2 Mikroservisna arhitektura

Mikroservisna arhitektura (*eng. microservice architecture*) nastoji riješiti navedene izazove razdvajanjem aplikacije na manje, samostalne servise koji se mogu neovisno razvijati, testirati, implementirati i skalirati. Ipak, s mikroservisima dolaze i novi izazovi, poput: složenosti upravljanja raspodijeljenim sustavima, komunikacije između servisa, sigurnosti i nadzora pojedinih komponenti sustava. Češće se mogu javljati problemi koji su povezani s mrežom, poput latencije, gubitka paketa i sl. Također, razvojni timovi moraju biti spremni na promjene u načinu rada, jer mikroservisna arhitektura zahtijeva drugačiji pristup razvoju, testiranju i implementaciji softvera.

Dizajn orijentiran na mikroservise (*eng. service-oriented design)* dobiva na popularnosti sredinom 2010-ih godina, kada su *early-adopteri* poput Netlifxa i Amazona počeli javno dijeliti svoja iskustva prijelazom s monolitne na mikroservisnu arhitekturu. 2015. . godine, James Lewis i Martin Fowler objavili su članak pod nazivom "Microservices" koji je postao jedan od najutjecajnijih izvora informacija o mikroservisnoj arhitekturi. U članku su definirali ključne karakteristike mikroservisne arhitekture i istaknuli prednosti koje ona donosi u odnosu na tradicionalne monolitne pristupe. Popularizacijom **Docker** i **Kubernetes** tehnologija, koje olakšavaju implementaciju i upravljanje mikroservisima, mikroservisna arhitektura postaje široko prihvaćena praksa u industriji softverskog razvoja.

Amazon je nekoliko svojih ključnih proizvoda, poput Amazon Primea, prebacio na mikroservisnu arhitekturu, dok je za neke druge proizvode zadržao monolitnu arhitekturu. Kroz vlastiti razvoj i uspon mikroservisnih tehnologija, Amazon prepoznaje poslovnu priliku u pružanju mikroservisne arhitekture kao usluge (*Microservice as a Service - MaaS*, spominje se i kao *Function as a Service - FaaS*) drugim tvrtkama, što rezultira razvojem platforme **AWS** (*Amazon Web Services*). Danas je AWS vodeći globalni pružatelj cloud usluga i jedan od najvećih izvora prihoda Amazon grupe. S druge strane, **Netflix** je potpuno migrirao na mikroservisnu arhitekturu i danas je jedan od najvećih korisnika AWS-a, ističući se kao primjer uspješne transformacije s monolitne na mikroservisnu arhitekturu. Navodi se da Netflix ima preko 1000 aktivnih

mikroservisa koji se izvršavaju u oblaku.



> Apstraktna ilustracija mikroservisne arhitekture Netflixa, izvor: zdnet.com

Važno je naglasiti da mikroservisna arhitektura nije univerzalno rješenje koje automatski otklanja sve izazove u razvoju kvalitetnog softvera. Ako se ne primjenjuje pažljivo i promišljeno, vrlo lako može dovesti do dodatne složenosti te stvoriti nove izazove u razvoju, održavanju i upravljanju sustavom, što naposljetku povećava potrebu za financijskim, vremenskim i ljudskim resursima. Zbog toga je ključno jasno razumjeti kada i na koji način primijeniti mikroservise, uzimajući u obzir specifične potrebe i kontekst projekta, veličinu i kompetencije razvojnog tima, raspoloživi budžet te kratkoročne i dugoročne ciljeve organizacije.

> Dobar Medium članak na ovu temu: When to Use and When NOT to Use Microservices: No Silver Bullet

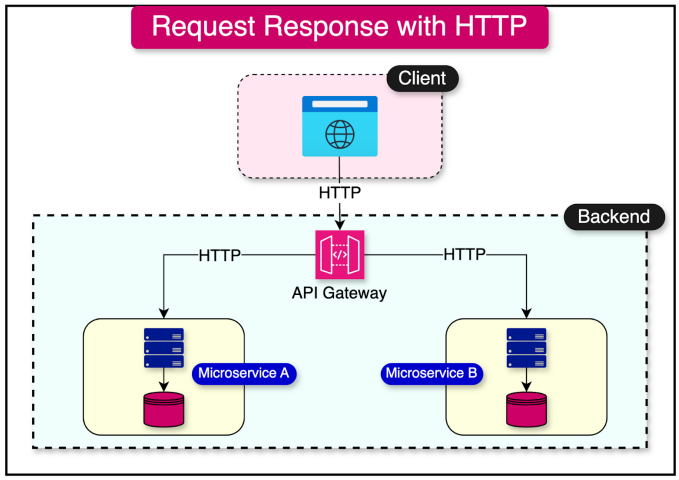
Core principi mikroservisne arhitekture

1. **Service Autonomy**: Each service is independently *deployable*, *testable*, *versioned* and *scalable*.
2. **Loose coupling and high cohesion**: Services should have minimal dependencies on each other and should be designed around specific business capabilities.
3. **Domain-Driven Design (DDD)**: Services should be modeled around business domains and bounded contexts.
4. **Api-first design:** Clear contracts should be established between services through well-defined APIs.
5. **Polyglot programming** : Different services can be implemented using different programming languages and technologies based on their specific requirements.

2. Definiranje poslužitelja koristeći `aiohttp`

U ovoj skripti dotaknuti ćemo se razvoja HTTP poslužitelja koristeći `aiohttp` biblioteku. `aiohttp` omogućuje razvoj *lightweight* asinkronih poslužitelja kojima možemo "otvoriti" naše poslovne aplikacije prema mreži te na taj način omogućiti komunikaciju između različitih servisa, odnosno čvorova u našem raspodijeljenom sustavu.

Mikroservis zamišljamo kao malu *lightweight* aplikaciju koja dobro i samostalno obavlja jednu specifičnu ili skup srodnih poslovnih funkcionalnosti koji ima smisla grupirati, a svoje usluge nudi putem nekog komunikacijskog protokola, odnosno u našem slučaju **HTTP poslužitelja**.



> Prikaz dva mikroservisa (A i B) s internim bazama podataka koji izlažu svoje funkcionalnosti putem HTTP poslužitelja definiranog preko AWS API Gateway servisa. Mikroservisi mogu i ne moraju koristiti HTTP protokol izlaganje svojih funkcionalnosti (ovisno o prirodi mikroservisa, komunikacija se može odvijati i drugim protokolima)

Na budućim vježbama bavit ćemo se drugim aspektima mikroservisne arhitekture, poput njihova testiranja, nadzora, skaliranja, kontejnerizacije i orkestracije. *Stay tuned!*

Nakon ovog teorijskog uvoda u mikroservisnu arhitekturu, u nastavku ove skripte ćemo se baviti implementacijom HTTP poslužitelja koji možemo otvarati prema mreži koristeći `aiohttp` biblioteku.

2.1 Ponavljanje: `aiohttp` klijentska sesija

Do sada smo koristili `aiohttp` biblioteku prvenstveno kroz `ClientSession` klasu za slanje asinkronih HTTP zahtjeva prema vanjskim servisima. Ovdje smo na neki način definirali HTTP klijenta unutar Pythona koji komunicira s vanjskim servisom - npr. CatFact ili JSONPlaceholder servisom u *Cloudu*.

Klijent je program ili komponenta koja inicira komunikaciju s poslužiteljem kako bi zatražio određene resurse ili usluge. U kontekstu web aplikacija, klijent je obično web preglednik ili aplikacija koja šalje HTTP zahtjeve prema web poslužitelju. U kontekstu mikroservise arhitekture, **klijent može biti bilo koji mikroservis koji šalje zahtjeve prema drugom mikroservisu** (eng. service-to-service communication).

Međutim, `aiohttp` je također odličan alat za izgradnju *lightweight* HTTP poslužitelja, direktno unutar Python aplikacije, to radimo kroz tzv. Server API.

> Zašto *lightweight*? Zato što `aiohttp` nije memorijski i implementacijski "težak" okvir poput npr. **Django**, **Flask** ili **FastAPI**. `aiohttp` pruža osnovne alate potrebne za razvoj asinkronih HTTP poslužitelja, ali ne dolazi s dodatnim slojevima apstrakcije ili funkcionalnostima koje su često prisutne u većim web okvirima. To znači da programer ima veću kontrolu nad implementacijom i može prilagoditi poslužitelj prema specifičnim potrebama aplikacije, ali također zahtijeva više ručnog rada i razumijevanja osnovnih principa web razvoja.

Prisjetimo se kako definirati **klijentsku sesiju** u `aiohttp`:

```
import asyncio
import aiohttp

async def main():
  async with aiohttp.ClientSession() as session: # Otvaramo HTTP klijentsku sesiju
    response = await session.get(URL)
    print(response.status) # 200
asyncio.run(main())
```

Rekli smo da koristimo *context manager* `with` kada radimo s resursima koji se moraju zatvoriti nakon upotrebe. U ovom slučaju, `ClientSession` je resurs koji se mora zatvoriti nakon što završimo s radom. Nakon toga, zaključili smo da je praktično pokrenuti glavnu korutinu pomoću `asyncio.run(main())`, a zatim unutar te korutine pozivati druge korutine koje obavljaju asinkrone operacije. Konkurentno slanje više zahtjeva i agregaciju rezultata možemo postići kroz `asyncio.gather()`, ili kreiranjem `asyncio.Task` objekata.

Primjer slanja 5 konkurentnih zahtjeva koristeći `asyncio.Task` i `asyncio.gather`:

```
import asyncio
import aiohttp

async def fetch_fact(session):
  print("Šaljem zahtjev...")
  rezultat = await session.get("https://catfact.ninja/fact")
  return (await rezultat.json())["fact"] # Deserijalizacija JSON odgovora

async def main():
  async with aiohttp.ClientSession() as session:
    cat_tasks = [asyncio.create_task(fetch_fact(session)) for _ in range(5)] #
Pohranjujemo Task objekte u listu
    facts = await asyncio.gather(*cat_tasks) # Listu raspakiravamo koristeći * operator,
čekamo na rezultat izvršavanja svih Taskova
    print(facts)

asyncio.run(main())
```

U nastavku ćemo vidjeti kako definirati **HTTP poslužitelj** koristeći `aiohttp` biblioteku.

2.2 `aiohttp.web` modul

Kako bi implementirali **poslužitelj** koristeći `aiohttp`, koristimo `aiohttp.web` modul. Ovaj modul pruža sve potrebne alate za definiranje ruta (*endpointa*), obradu zahtjeva i slanje odgovora kroz HTTP protokol.

U kontekstu mikroservisne arhitekture, poslužitelj je komponenta koja zaprima zahtjeve od drugih mikroservisa, obrađuje ih i vraća odgovore. Zbog prirode mikroservisa, svaki mikroservis trebao bi imati neki oblik vlastitog sučelja koji otvara vrata njegovih funkcionalnosti prema vanjskom svijetu. Bilo da se radi o HTTP (*Express.js, FastAPI, aiohttp, Spring Boot*), gRPC ili nekom drugom, poslužitelj je ključna komponenta koja omogućava komunikaciju između različitih mikroservisa.

> Zapamti: Možemo zamisliti poslužitelj kao **tehničku implementaciju komunikacijskog sučelja mikroservisa**, dok je mikroservis sama konceptualna jedinica koja obavlja određenu poslovnu (ili strogo tehničku) funkcionalnost.

Modul nije potrebno naknadno instalirati, već je uključen u `aiohttp` paketu.

```
from aiohttp import web
```

Ključna komponenta `aiohttp.web` modula je `Application` klasa, koja definira glavnu aplikaciju (**poslužitelj**).

```
app = web.Application() # u varijablu app pohranjujemo instancu Application klase
```

Da bi pokrenuli poslužitelj, nije dovoljno samo pokrenuti Python skriptu, već moramo definirati na kojoj **adresi** i **portu** će poslužitelj slušati HTTP dolazne zahtjeve.

Poslužitelj pokrećemo pozivom metode `web.run_app()`:

Sintaksa:

```
web.run_app(app, host, port)
```

- `app` - instanca `Application` klase koju želimo pokrenuti
- `host` - adresa na kojoj će poslužitelj slušati (default: `'localhost'`)
- `port` - port na kojem će poslužitelj slušati (npr. `8080`)

Primjer pokretanja poslužitelja na adresi `localhost` i portu `8080`:

```
from aiohttp import web

app = web.Application()

web.run_app(app, host='localhost', port=8080)

# ili kraće
web.run_app(app, port=8080)
```

> Podsjetnik: `localhost` je posebna mrežna adresa koja se koristi za usmjeravanje prometa natrag na isti uređaj s kojeg je zahtjev poslan. To znači da kada aplikacija ili usluga koristi `localhost`, ona "komunicira sa sobom", tj. s istim računalom na kojem se izvršava. `localhost` se obično prevodi na IP adresu `127.0.0.1` za IPv4 ili `::1` za IPv6.

Ako je sve ispravno konfigurirano, poslužitelj će se pokrenuti i vidjet ćete ispis u terminalu:

```
======== Running on http://localhost:8080 ========
(Press CTRL+C to quit)
```

Možete otvoriti web preglednik i posjetiti adresu `http://localhost:8080` kako biste provjerili je li poslužitelj uspješno pokrenut ili poslati zahtjev koristeći neki od HTTP klijenata.

> Za **HTTP klijent unutar terminala** preporuka je koristiti curl.
> Kao **Desktop** ili **Web aplikaciju** preporuka je koristiti Postman ili Insomnia, međutim ima ih još mnogo.
> Praktično je i preporuka koristiti neku od **VS Code HTTP klijent ekstenzija**, primjerice Thunder Client ili REST Client.

Koristeći jedan od alata, pošaljite zahtjev na adresu `http://localhost:8080` i provjerite je li poslužitelj uspješno pokrenut.

2.3 Definiranje HTTP poslužiteljskih ruta (endpoints)

Da bi poslužitelj bio funkcionalan i mogao obrađivati dolazne zahtjeve, potrebno je definirati rute (*eng. route/endpoint*) koje će poslužitelj opsluživati (*eng. serve*). **Ruta** predstavlja URL putanju putem koje se pristupa određenom **resursu** ili funkcionalnosti.

Ako još niste, preporučuje se da se prisjetite osnova HTTP protokola (skripta RS4) kako biste bolje razumjeli ostatak skripte.

2.3.1 HTTP GET ruta

U `aiohttp.web` modulu, rute možete definirati na više načina. Primjerice, ako želite dodati jednostavnu GET rutu koja predstavlja HTTP zahtjev s GET metodom, koristite metodu `add_get()` na objektu `router`:

```
app.router.add_get(path, handler_function) # Dodajemo GET rutu na određenu putanju
```

- `path` - URL putanja na koju će se ruta primjenjivati (npr. `'/'`, `'/korisnici'`, `'/proizvodi'`)
- `handler_function` - funkcija koja će se pozvati kada se zahtjev uputi na određenu rutu

Handler funkcija (U JavaScriptu ekvivalent je *callback* funkcija) je funkcija koja će se izvršiti kada se zahtjev uputi na definiranu rutu. *Handler* funkcija može biti **sinkrona** ili **asinkrona** (**korutina**), međutim u praksi je preporučljivo koristiti asinkrone funkcije kako bi se izbjeglo blokiranje glavne dretve.

Handler funkcija prima **ulazni parametar** `request` koji predstavlja HTTP zahtjev koji je klijent napravio prema poslužitelju. Ovaj Python objekt sadrži sve informacije o HTTP zahtjevu, poput: URL putanje, HTTP metode, zaglavlja, tijela zahtjeva i sl.

```
def handler_function(request): # Sinkrona handler funkcija koja prima request objekt
  pass # Placeholder za implementaciju
```

Prikazat ćemo uobičajene podatke o zahtjevu koji su pohranjeni unutar objekta `request`:

```
from aiohttp import web

def handler_function(request):
  print(request.method) # HTTP metoda dolaznog zahtjeva
  print(request.path) # HTTP putanja (URL) na koju je zahtjev upućen
  print(request.headers) # HTTP zaglavlja dolaznog zahtjeva

app = web.Application()

app.router.add_get('/', handler_function) # Čitaj: Dodajemo GET rutu na putanju '/' koja poziva handler funkciju

web.run_app(app, host='localhost', port=8080)
```

> Podsjetnik: Putanju `'/'` nazivamo i *root* ili korijenskom putanjom poslužitelja.

Ispisuje: GET metodu, URL putanju (`/`), zaglavlja zahtjeva, te **klijenta s kojeg je zahtjev poslan** - u našem slučaju `curl`:

```
GET
/
<CIMultiDictProxy('Host': '0.0.0.0:8080', 'User-Agent': 'curl/8.7.1', 'Accept': '*/*')>
```

Vidjet ćete da smo uz ispis dobili i grešku. To je zato jer **nismo poslali HTTP odgovor natrag klijentu**. Ukoliko *handler* funkcija ne vrati odgovor, poslužitelj će vratiti grešku `500 Internal Server Error`. Da bismo to ispravili, moramo vratiti odgovor koristeći `web.Response` objekt:

```
def handler_function(request):
  return web.Response() # Vraćamo prazan HTTP odgovor
```

Nema više greške! Međutim, odgovor je prazan. Klasa `web.Response` omogućava nam da precizno definiramo HTTP odgovor koji će poslužitelj vratiti klijentu. Na primjer, možemo postaviti statusni kôd, zaglavlja i tijelo odgovora.

Sintaksa `web.Response` klasnog konstruktora:

```
aiohttp.web.Response(
    body=None,
    status=200,
    reason=None,
    text=None,
    headers=None,
    content_type=None,
    charset=None
)
```

- `body` - tijelo odgovora (npr. `HTML`, JSON)
- `status` - statusni kôd odgovora (npr. `200`, `404`, `500`)

- `reason` - tekstualni opis statusnog kôda (npr. `'OK'`, `'Not Found'`, `'Internal Server Error'`)
- `text` - tekstualno tijelo odgovora (npr. `'Hello, world!'`)
- `headers` - zaglavlja odgovora (npr. `{'Content-Type': 'application/json'}`)
- `content_type` - oblik sadržaja odgovora (npr. `'text/html'`, `'application/json'`)
- `charset` - karakterna enkodiranje odgovora (gotovo uvijek: `'utf-8'`)

Primjer vraćanja jednostavnog HTML odgovora koji vraća tekst `'Pozdrav Raspodijeljeni sustavi!'`:

```
def handler_function(request):
  return web.Response(text='Pozdrav Raspodijeljeni sustavi!')
```

- Otvorite web preglednik i posjetite adresu `http://localhost:8080` kako biste vidjeli rezultat, odnosno pošaljite zahtjev koristeći HTTP klijent.

Pomoću naredbe `curl` možete poslati HTTP zahtjev direktno iz terminala:

```
→ curl http://localhost:8080

# ili s eksplicitnim naglašavanjem HTTP metode opcijom/zastavicom "X"

→ curl -X GET http://localhost:8080
```

Nakon svake promjene u kôdu poslužitelja potrebno je ponovno pokrenuti skriptu kako bi se promjene primijenile. To je zato što jednom kad se skripta pokrene, unutar terminala se pokreće proces koji sluša na definiranoj adresi i portu. Svakom izmjenom poslužitelja, potrebno je prekinuti trenutačni proces (npr. pritiskom `Ctrl/CMD + C`) i ponovno pokrenuti skriptu.

2.3.2 Automatsko ponovno pokretanje poslužitelja (hot/live reloading)

Tijekom razvoja, ovo brzo postaje nepraktično i zamorno, pa je topla preporuka instalirati jedan od alata koji omogućuju **automatsko ponovno pokretanje poslužitelja nakon promjena u kôdu**, tzv. *hot/live reloading*.

U tu svrhu, možete instalirati neki od sljedećih alata:

1. Nodemon - prvenstveno za Node.js aplikacije, ali može se koristiti i za Python. Nodemon se instalira u globalnom okruženju (flag `-g`) i pokreće se iz terminala.

```
→ npm install -g nodemon # slobodno instalirajte iz bilo koje terminal sesije
```

- ako ne radi, provjerite je li dodan u PATH globalnu varijablu i ponovno pokrenite VS Code/terminal
- naravno, potrebno je instalirati i Node.js runtime.

Pokretanje:

```
→ nodemon --exec python index.py
```

2. aiohttp-devtools - specifično za `aiohttp` aplikacije. Instalacija:

```
→ pip install aiohttp-devtools
```

Pokretanje:

```
→ adev runserver index.py
```

3. watchdog - generalni alat za praćenje promjena u datotekama. Kompleksniji za postavljanje budući da je, osim instalacije, potrebno napisati skriptu koja će pokrenuti poslužitelj.

> Preporuka je koristiti `aiohttp-devtools` ili `nodemon` jer su jednostavniji za postavljanje i korištenje.

2.3.3 HTTP GET - slanje JSON odgovora

Jednom kad ste uspješno podesili *hot-reload* funkcionalnost, možemo se vratiti na razvoj poslužitelja. U praksi, često ćete (gotovo uvijek) se susresti s potrebom slanja JSON odgovora iz poslužitelja, budući da je JSON format postao de *facto standard* za razmjenu podataka između web servisa.

Rekli smo da format HTTP odgovora možemo definirati kroz `web.Response` objekt:

```
def handler_function(request):
  return web.Response(text='Pozdrav Raspodijeljeni sustavi!') # Ovo vraća tekstualni
odgovor
```

Ako želimo poslati JSON odgovor, stvari su nešto kompliciranije jer moramo odraditi serijalizaciju podataka u JSON format prije samog slanja.

Podsjetnik (u grubo):

- **Serijalizacija** - pretvaranje Python objekta u JSON format
- **Deserijalizacija** - pretvaranje JSON formata u Python kolekciju (*npr. list, dict, str*)

Za pretvaranja Python objekta u JSON format, možemo upotrijebiti ugrađeni modul `json`:

Za serijalizaciju koristimo metodu `json.dumps()`:

```
import json

data = {'ime': 'Ivo', 'prezime': 'Ivić', 'godine': 25}

json_data = json.dumps(data) # Serijalizacija Python objekta u JSON string

# JSON format je tipa string
print(type(json_data)) # <class 'str'>
```

U `web.Response` **moramo precizirati da se radi o JSON formatu** kako bi klijent znao kako interpretirati odgovor.

- to radimo kroz parametar `content_type`:

```
def handler_function(request):
  data = {'ime': 'Ivo', 'prezime': 'Ivić', 'godine': 25}
  return web.Response(text=json.dumps(data), content_type='application/json')
```

Drugi i preporučeni način je korištenje metode `json_response()` koja **automatski serijalizira Python objekt u JSON format** prilikom slanja odgovora:

```
def handler_function(request):
  data = {'ime': 'Ivo', 'prezime': 'Ivić', 'godine': 25}
  return web.json_response(data) # Automatska serijalizacija u JSON format, preporučeno
```

Ovdje ne koristimo generičku `web.Response` klasu, već specijaliziranu `web.json_response()` funkciju koja automatski serijalizira Python objekt u JSON format i **postavlja odgovarajuće zaglavlje HTTP odgovora (`Content-Type: application/json`)**.

> U praksi, preporučuje se koristiti `web.json_response()` funkciju jer je kôd kraći i čitljiviji

---

Rezime:

Do sad smo definirali sljedeće dijelove `aiohttp` poslužitelja:

1. `Application` instanca koja predstavlja glavnu aplikaciju

```
app = web.Application()

web.run_app(app, port=8080) # Pokretanje poslužitelja
```

2. GET ruta na putanju `'/'` koja poziva *handler* funkciju

```
app.router.add_get(path, handler_function)
```

3. *handler* funkcija koja obrađuje zahtjev i vraća odgovor, može biti sinkrona ili asinkrona (korutina)

```
def handler_function(request):
  return web.json_response(data) # Automatska serijalizacija u JSON format

def handler_function_dva(request):
  return web.Response(text='Pozdrav Raspodijeljeni sustavi!') # Vraćanje tekstualnog
odgovora kroz standardni web.Response objekt
```

2.3.4 HTTP POST ruta

Za razliku od GET metode koja se koristi za dohvaćanje podataka, **POST metoda** se prvenstveno koristi za **slanje podataka prema poslužitelju**.

Kod web aplikacija, podaci koji se šalju POST metodom najčešće su iz forme koju je korisnik popunio.

Na primjer: prilikom registracije korisnika, unos korisničkog imena, lozinke i e-mail adrese šalje se prema poslužitelju POST metodom. Takvi podaci se danas najčešće šalju u JSON formatu.

Kod poslužitelja mikroservisa, POST metoda i srodne metode (PUT, PATCH, DELETE) mogu se koristiti za razmjenu podataka između različitih mikroservisa.

Primjer: `servis_1` može poslati POST zahtjev prema `servis_2` kako bi zatražio provedbu plaćanja, pri čemu `servis_1` šalje podatke o transakciji u JSON formatu unutar tijela HTTP zahtjeva.

U `aiohttp.web` modulu, POST rutu definiramo kroz metodu `add_post()` na objektu `router`:

```
app.router.add_post(path, handler_function)
```

***Handler* funkcija** koja obrađuje POST zahtjev prima dodatni parametar `request` jednako kao kod GET metode. Međutim, POST metoda omogućava pristup tijelu zahtjeva (eng. *request body*) koje sadrži podatke koje je klijent poslao prema poslužitelju.

> U nastavku ćemo *handler* funkcije definirati kao **korutine** kako bismo mogli asinkrono obrađivati HTTP zahtjeve.

Deserijalizaciju podataka iz JSON formata u Python objekt možemo obaviti kroz metodu `json()` objekta `request`, na isti način kao što smo to radili prilikom slanja zahtjeva prema vanjskim servisima kod klijentske sesije.

> Uočite: ne koristimo ugrađeni `json` modul kao kod serijalizacije, već **metodu** `.json()` objekta `Request`.

Sintaksa:

```
data = await request.json()
```

Primjer definiranja POST rute koja prima JSON podatke i vraća odgovor:

```
from aiohttp import web

async def post_handler(request):
  data = await request.json() # Deserijalizacija JSON podataka
  print(data) # Ispis podataka u terminal
  return web.json_response(data) # Vraćanje istih podataka kao odgovor

app = web.Application()

app.router.add_post('/', post_handler) # Dodajemo POST rutu na putanju '/' koja poziva post_handler korutinu

web.run_app(app, host='localhost', port=8080)
```

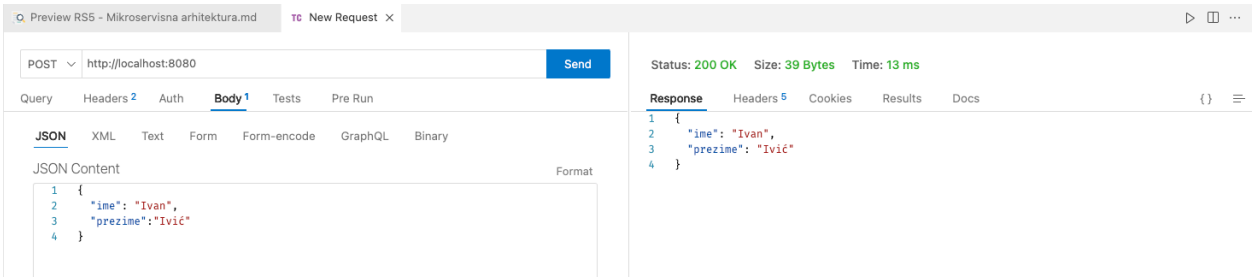
Podatke pošaljite kroz neki od **HTTP klijenata** ili `curl` (`-H` opcija za postavljanje zaglavlja, `-d` opcija za definiranje HTTP tijela):

```
→ curl -X POST -H "Content-Type: application/json" -d '{"ime": "Ivo", "prezime": "Ivić",
"godine": 25}' http://localhost:8080
```

Očekivani odgovor (isti podaci kao u zahtjevu):

```
{'ime': 'Ivo', 'prezime': 'Ivić', 'godine': 25}
```

Puno jednostavnije je slanje kroz HTTP klijent jer ne moramo eksplicitno navoditi zaglavlja u zahtjevu (Postman, Insomnia, Thunder Client, REST Client i slični alati to rade za nas).



> Primjer slanja POST zahtjeva s JSON tijelom na `http://localhost:8080` kroz Thunder Client VS Code ekstenziju

---

Za dodavanje preostalih HTTP metoda (PUT, DELETE, PATCH) koristimo odgovarajuće ekvivalente na objektu `router`:

- `router.add_put()` - dodavanje PUT rute
- `router.add_patch()` - dodavanje PATCH rute
- `router.add_delete()` - dodavanje DELETE rute

Ali možemo koristiti i generičku metodu `router.add_routes()` koja prima **listu ruta koje želimo dodati**.

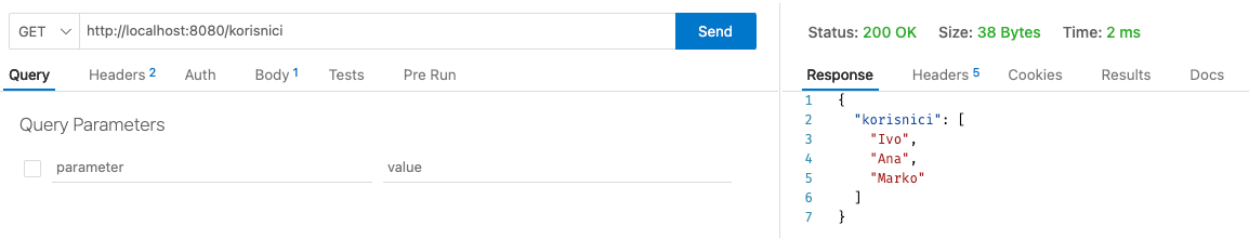
Sintaksa:

```
app.router.add_routes([
  web.get(path, handler_function), # Dodavanje GET rute
  web.post(path, handler_function), # Dodavanje POST rute
  web.put(path, handler_function), # Dodavanje PUT rute
  web.delete(path, handler_function) # Dodavanje DELETE rute
  ... # itd.
])
```

Primjer, definirat ćemo poslužitelj s dvije rute: `GET /korisnici` i `POST /korisnici`:

```
from aiohttp import web

async def get_users(request): # korutina za GET zahtjev
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko']})

async def add_users(request): # korutina za POST zahtjev
  data = await request.json()
```

```
  return web.json_response(data) # Vraćamo isti podatak, bez ikakve obrade

app = web.Application()

app.router.add_routes([
  web.get('/korisnici', get_users), # GET /korisnici
  web.post('/korisnici', add_users) # POST /korisnici
])

web.run_app(app, port=8080)
```



> Primjer slanja GET zahtjeva na `http://localhost:8080/korisnici` kroz Thunder Client ekstenziju; odgovor je lista korisnika

2.4 Zadaci za vježbu: Definiranje jednostavnih `aiohttp` poslužitelja

Zadatak 1: `GET /proizvodi`

Definirajte `aiohttp` poslužitelj koji radi na portu `8081` koji na putanji `/proizvodi` vraća listu proizvoda u JSON formatu. Svaki proizvod je rječnik koji sadrži ključeve `naziv`, `cijena` i `količina`. Pošaljite zahtjev na adresu `http://localhost:8081/proizvodi` koristeći neki od HTTP klijenata ili `curl` i provjerite odgovor.

Zadatak 2: `POST /proizvodi`

Nadogradite poslužitelj iz prethodnog zadatka na način da na istoj putanji `/proizvodi` prima POST zahtjeve s podacima o proizvodu. Podaci se šalju u JSON formatu i sadrže ključeve `naziv`, `cijena` i `količina`. *Handler* funkcija treba ispisati primljene podatke u terminalu, dodati novi proizvod u listu proizvoda i vratiti **odgovor s novom listom proizvoda** u JSON formatu.

Zadatak 3: `GET /punoljetni`

Definirajte poslužitelj koji sluša na portu `8082` i na putanji `/punoljetni` vraća listu korisnika starijih od 18 godina. Svaki korisnik je rječnik koji sadrži ključeve `ime` i `godine`. Pošaljite zahtjev na adresu `http://localhost:8082/punoljetni` i provjerite odgovor. Novu listu korisnika definirajte koristeći funkciju `filter` ili `list comprehension`.

```
korisnici = [
  {'ime': 'Ivo', 'godine': 25},
  {'ime': 'Ana', 'godine': 17},
  {'ime': 'Marko', 'godine': 19},
  {'ime': 'Maja', 'godine': 16},
  {'ime': 'Iva', 'godine': 22}
]
```

3. Klijent-Poslužitelj komunikacija koristeći `aiohttp` biblioteku

Klijent-poslužitelj (eng. *client-server*) arhitektura je komunikacijski model u kojem klijent (npr. web preglednik ili aplikacija) šalje zahtjeve prema poslužitelju (npr. web serveru) koji obrađuje te zahtjeve i vraća odgovore natrag klijentu.

U prethodnom poglavlju smo definirali `aiohttp` pozlužitelj koji sluša na definiranoj adresi i portu te obrađuje dolazne zahtjeve, dok smo u skripti RS4 vidjeli kako se koristi `aiohttp` klijentska sesija za slanje asinkronih i konkurentnih HTTP zahtjeva koristeći `ClientSession` klasu.

U ovom dijelu ćemo spojiti ta dva koncepta i pokazati **kako unutar Python kôda možemo simulirati komunikaciju između klijenta i poslužitelja** koristeći `aiohttp` klijentsku sesiju i poslužitelj definiran kroz `aiohttp.web` modul.

> **Zašto bismo ovo radili**? Mikroservise koje gradimo na ovom kolegiju ćemo kroz lokalno razvojno okruženje (`localhost`) izlagati putem `aiohttp` poslužitelja. Međutim, kako mikroservisna arhitektura nije klasični *klijent-poslužitelj* komunikacijski model, već je ***service-to-service*** komunikacija, jako često ćemo imati situaciju gdje mikroservis šalje zahtjeve prema drugom mikroservisu. U tom slučaju, mikroservis koji šalje zahtjev "ponaša se kao klijent", dok se mikroservis koji prima zahtjev i obrađuje ga "ponaša kao poslužitelj".

Krenut ćemo od definicije jednostavnog poslužitelja koji sluša na adresi `localhost` i portu `8080` te na putanji `/korisnici` vraća listu korisnika u JSON formatu:

```
from aiohttp import web

async def get_users(request):
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva', 'Ivan']})
#hardkodirani podaci - stvarni će bit u bazi podataka

app = web.Application()

app.router.add_get('/korisnici', get_users)

web.run_app(app, host='localhost', port=8080)
```

Klijentsku sesiju smo dosad otvarali unutar `main` korutine koristeći *context manager* pa ćemo to i ovdje učiniti:

```
import asyncio

async def main():
  async with aiohttp.ClientSession() as session:
    pass

asyncio.run(main())
```

Ako spojimo kôd, dobivamo sljedeće:

```
from aiohttp import web
import asyncio, aiohttp

async def get_users(request):
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva', 'Ivan']})

# Definicija poslužitelja
app = web.Application()

app.router.add_get('/korisnici', get_users)
# Pokretanje poslužitelja
web.run_app(app, host='localhost', port=8080)

# main korutina za klijentsku sesiju
async def main():
  async with aiohttp.ClientSession() as session:
    print("Klijentska sesija otvorena")
asyncio.run(main())
```

Koji problem uočavate?

▸ Spoiler alert! Odgovor na pitanje

Problem je što **ako pokrenemo poslužitelj, on će blokirati izvođenje ostatka kôda**, uključujući otvaranje klijentske sesije. Ovo je zato što funkcija `web.run_app()` blokira izvršavanje ostatka kôda **okupuacijom glavne dretve procesa** sve dok poslužitelj radi.

> Zamislite da naš mikroservis s aktivnim poslužiteljem želi poslati zahtjev prema drugom mikroservisu - ne želimo prekinuti rad poslužitelja da bismo poslali zahtjev, već želimo da poslužitelj i klijentska sesija rade istovremeno (**konkurentno**).
> *Primjer:* `microservice_1` ima otvoreni poslužitelj koji prima zahtjeve od `microservice_2`, međutim, primitkom zahtjeva, `microservice_1` treba poslati zahtjeve prema `microservice_3` koji obrađuje neku treću funkcionalnost. **Prekid rada poslužitelja `microservice_1` (kako bi poslao zahtjeve) nije prihvatljiva opcija**, budući da za vrijeme prekida poslužitelj ne može obrađivati potencijalne dolazne zahtjeve od `microservice_2`.
> Opisano je na neki način nedostatak čistog HTTP klijent-poslužitelj modela u kontekstu mikroservisne arhitekture.

Idemo pokušati ovo riješiti. U Pythonu možemo iskoristiti specijalnu varijablu `__name__`, koja uvijek sadrži naziv trenutnog modula. Kada skriptu pokrenemo direktno, vrijednost `__name__` bit će `__main__`. S druge strane, ako skriptu uvezemo u neki drugi modul, `__name__` će sadržavati naziv tog modula.

Korištenjem uvjetnog izraza `if __name__ == '__main__':` možemo definirati blok kôda koji će se izvršiti samo ako skriptu pokrenemo direktno, a neće se izvršiti ako je uvezemo kao modul u neki drugi kôd.

Sintaksa:

```
if __name__ == '__main__':
  # Blok kôda koji se izvršava samo ako skriptu pokrenemo direktno (npr. python index.py)
```

- isto će raditi za pokretanje kroz `nodemon` ili `aiohttp-devtools`

Primjerice, možemo premjestiti pokretanje poslužitelja unutar ovog bloka:

```
if __name__ == '__main__':
  print("Pokrećem samo poslužitelj")
  web.run_app(app, host='localhost', port=8080)
```

Hoćemo li sada pokrenuti klijentsku sesiju i poslužitelj zajedno?

Primjer ukupnog kôda:

```
from aiohttp import web
import asyncio, aiohttp

async def get_users(request):
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva', 'Ivan']})

app = web.Application()

app.router.add_get('/korisnici', get_users)

async def main():
  async with aiohttp.ClientSession() as session:
    print("Klijentska sesija otvorena")

asyncio.run(main()) # pokreće klijentsku sesiju

if __name__ == '__main__':
  print("Direktno pokrenuta skripta...")
  web.run_app(app, host='localhost', port=8080) # pokreće poslužitelj
```

Kôd iznad će svakako prvo otvoriti klijentsku sesiju, obzirom da se `asyncio.run` poziva prije pokretanja poslužitelja. Ako ne želimo pokrenuti poslužitelj, možemo samo zakomentirati liniju `web.run_app(app, host='localhost', port=8080)`.

Međutim je li moguće na ovaj način pokrenuti poslužitelj, **a nakon toga** pozvati `main` korutinu koja otvara klijentsku sesiju? Više nam ima smisla prvo pokrenuti poslužitelj, a zatim otvoriti klijentsku sesiju koja će slati zahtjeve prema tom poslužitelju (**ili u praksi - drugom mikroservisu**).

Premjestit ćemo `asyncio.run(main())` unutar bloka `if __name__ == '__main__':`:

```
from aiohttp import web
import asyncio, aiohttp

async def get_users(request):
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva', 'Ivan']})

app = web.Application()

app.router.add_get('/korisnici', get_users)

async def main():
  async with aiohttp.ClientSession() as session:
    print("Klijentska sesija otvorena")
    pass

if __name__ == '__main__':
  print("Direktno pokrenuta skripta...")
  web.run_app(app, host='localhost', port=8080) # pokreće poslužitelj
  asyncio.run(main()) # hoće li se pokrenuti?
```

▸ Spoiler alert! Odgovor na pitanje

Kako bismo onda riješili ovaj problem? Potrebno je koristiti drugačiji pristup za pokretanje poslužitelja koji **ne blokira izvršavanje ostatka kôda**, odnosno **potrebno je pokrenuti poslužitelj i druge asinkrone operacije** unutar istog *event loopa*. Na ovaj način možemo **konkurentno izvršavati zadatke** na razini poslužitelja (*server-tasks*) i klijentske sesije (*client-tasks*).

3.1 Izvršavanje pozadinske korutine s poslužiteljem

Na primjeru iznad vidjeli smo da `web.run_app()` blokira izvršavanje ostatka kôda, međutim, što ako moramo implementirati da naš mikroservis održava pozadinski zadatak koji se izvršava periodično, čeka na događaje ili obrađuje poruke iz reda poruka (*eng. message queue*)? U tom slučaju, trebamo način da pokrenemo poslužitelj i istovremeno izvršavamo druge asinkrone zadatke unutar istog *event loopa* - oblik *background processinga*.

Primjer: Mikroservis koji ima definirani HTTP poslužitelj i istovremene obrađuje poruke iz *mock RabbitMQ* reda poruka.

> Napomena: RabbitMQ je popularni sustav za razmjenu poruka koji omogućuje mikroservisima da komuniciraju asinkrono putem slanja i primanja poruka (eng. *message broker*). Radi se o sustavu baziranom na *Advanced Message Queuing Protocol* (AMQP) standardu koji omogućuje pouzdanu i skalabilnu razmjenu poruka između različitih komponenti. Aplikacija koja šalje poruke naziva se *producer*, dok aplikacija koja prima i obrađuje poruke naziva se *consumer*. RabbitMQ omogućuje kreiranje redova poruka (eng. *message queues*) gdje se poruke pohranjuju dok ih *consumeri* ne preuzmu i obrade.
> U ovom primjeru nećemo implementirati stvarnu integraciju s RabbitMQ-om, već ćemo simulirati pozadinsku obradu poruka kroz jednostavnu korutinu koja periodično ispisuje da obrađuje pristigle poruke (nije *event-driven* niti asinkrono, već obična simulacija pozadinskog zadatka koji postoji).

```
from aiohttp import web
import asyncio, aiohttp

async def get_users(request):
  return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva', 'Ivan']})

app = web.Application()

app.router.add_get('/korisnici', get_users)

async def process_messages():
  while True:
    print("Obrađujem poruke iz reda...") # Ovdje bi išla logika za obradu poruka iz
RabbitMQ reda
    await asyncio.sleep(5) # Simuliramo čekanje na nove poruke

async def main():
  await process_messages() # pokretanje pozadinskog process_messages zadatka

if __name__ == '__main__':
  asyncio.run(main()) # pokretanje glavne korutine
  web.run_app(app, host='localhost', port=8080) # pokretanje poslužitelja (hoće li se
pokrenuti?)
```

Poslužitelj na mikroservisu iznad neće se pokrenuti jer će se izvršenje kôda blokirati na liniji `asyncio.run(main())`, tj. pokrenuti će se *event loop* koji izvodi *blocking* `process_messages` korutinu i na taj način **nikada neće doći do linije** `web.run_app(...)`.

Što ako bismo pokušali pokrenuti korutine konkurentno, koristeći `asyncio.create_task()` unutar `main` korutine? Ideja je da pokrenemo `process_messages` kao pozadinski zadatak na način da ga rasporedimo prvo unutar *event loopa*, a zatim pokrenemo poslužitelj.

```
async def main():
  asyncio.create_task(process_messages()) # pokretanje pozadinskog process_messages
zadatka raspoređivanjem unutar event loopa
  print(asyncio.get_running_loop().is_running()) # Ispisuje: True - event loop je pokrenut
i radi
  web.run_app(app, host='localhost', port=8080) # pokušat će pokrenuti novi event loop
(GREŠKA!)

if __name__ == '__main__':
  asyncio.run(main()) # pokretanje glavne korutine
```

Ovo također neće raditi, ali iz nešto drugačijeg razloga. Ovaj kôd nastojat će pokrenuti dva *event loopa* unutar istog procesa: jedan kroz `asyncio.run(main())` i drugi kroz `web.run_app()`, što nije dozvoljeno u jednodretvneom Python procesu. Razlog ovome je što `web.run_app()` interno poziva `asyncio.run()`, što znači da pokušavamo pokrenuti novi *event loop* dok je već jedan aktivan.

```
RuntimeError: Cannot run the event loop while another loop is running
Task was destroyed but it is pending!
```

> **Zapamtite**: Mikroservisi često trebaju održavati pozadinske zadatke dok istovremeno služe zahtjeve putem poslužitelja. Pozadinski zadaci mogu uključivati obradu poruka iz redova poruka, periodične zadatke ili druge asinkrone operacije koje ne smiju blokirati glavni tok izvršavanja poslužitelja. Također, mikroservisi često mogu pokretati mini-izolirana radna okruženja unutar istog procesa, što zahtijeva fleksibilnost u upravljanju *event loopom* kako ne bi došlo do sukoba između različitih komponenti ili *race-condition* situacija.

> Stvarni *production-ready* mikroservisi velikih IT tvrtki, deployani u cloudu i dostupni za najam, često nude **više programskih sučelja putem kojih ih korisnici mogu koristiti**. Primjerice, AWS mikroservisi često imaju REST i SDK sučelja, što omogućava jednostavnu integraciju u različite aplikacije i sisteme.

Race-condition problem

Race-condition problem predstavlja softversku grešku gdje rezultat (ishod) neke operacije ovisi o nizu nepredvidivih događaja, poput redoslijeda izvršavanja dretvi/procesa ili korutina i promjenu stanja zajedničkih resursa. Kod mikroservisa, **ovaj se problem često javlja** kada više komponenti pokušava istovremeno pristupiti ili mijenjati zajedničke resurse, poput baze podataka, datoteka ili mrežnih veza, bez odgovarajuće sinkronizacije i česti je *challenge* prilikom razvoja raspodijeljenih sustavima.

Najčešći razlozi za pojavu *race-condition* problema su:

- **priroda konkurentnog izvršavanja**: više dretvi/procesa/korutina istovremeno pristupa zajedničkim resursima
- **dijeljeni resursi bez odgovarajuće sinkronizacije** (npr. stanje na bankovnom računu, računalna datoteka, stanje varijable u memoriji, web poslužitelji)
- **nepredvidivi redoslijed izvršavanja dretvi/procesa/korutina**
- **programi se "natječu" s izvršenjem njihovih operacija**, a konačno stanje se može razlikovati ovisno o tome koja je dretva/proces/korutina prva završila

3.2 `AppRunner` klasa - konkurentno pokretanje poslužitelja unutar postojećeg event loopa

AppRunner klasu koristimo kada nam je potrebna veća kontrola nad životnim ciklusom poslužitelja, primjerice kada želimo pokrenuti poslužitelj unutar aktivnog *event loopa*, istovremeno pokrenuti više poslužitelja na različitim adresama ili ih pokrenuti na različitim mrežnim sučeljima (*eng. network interfaces*).

Prednost `AppRunner` klase je što, za razliku od funkcije `web.run_app()`, **ne blokira izvršavanje glavne dretve**, odnosno omogućuje *non-blocking* pokretanje poslužitelja. U tom slučaju, *event loop* može nastaviti istovremeno izvršavati druge asinkrone zadatke, poput otvaranja djelomičnih klijentskih sesija, obrade poruka iz redova poruka ili drugih periodičkih *cron-like* zadataka - npr. slanje izvještaja o stanju mikroservisa na određeni interval ili provjera zdravlja povezanih servisa.

`AppRunner` se obično koristi zajedno s `TCPSite`, što omogućuje povezivanje poslužitelja s određenim mrežnim sučeljem i portom.

`AppRunner` klasu uključujemo iz `aiohttp.web` modula:

```
from aiohttp.web import AppRunner
```

Da bismo pokrenuli poslužitelj koristeći `AppRunner`, prvo kreiramo instancu klase i pripremimo je za pokretanje:

```
runner = AppRunner(app)
```

Postupak je sljedeći:

1. Definiraj AppRunner instancu
2. Postavi AppRunner instancu pozivom `await runner.setup()`
3. Poveži poslužitelj s mrežnim sučeljem i portom kreiranjem TCPSite instance
4. Pozovi `await site.start()` kako bi se poslužitelj pokrenuo

Sintaksa:

```
runner = AppRunner(app)
await runner.setup()
site = TCPSite(runner, host, port)
await site.start()
```

Primjer za lokalni poslužitelj na portu `8080`:

```
from aiohttp.web import AppRunner, TCPSite

runner = AppRunner(app)          # 1. Definiraj AppRunner instancu
await runner.setup()             # 2. Postavi AppRunner instancu
site = TCPSite(runner, 'localhost', 8080)  # 3. Poveži poslužitelj s localhost:8080
await site.start()               # 4. Pokreni poslužitelj
```

Ova četiri koraka često se ponavljaju, pa ih je **praktično spakirati u zasebnu korutinu**, npr. `start_server`:

```
async def start_server():
    runner = AppRunner(app)
    await runner.setup()
    site = TCPSite(runner, "localhost", 8080)
    await site.start()
    print("Poslužitelj sluša na http://localhost:8080")

await start_server()  # Hoće li se pokrenuti?
```

Ako želimo pokrenuti poslužitelj i istovremeno izvršavati druge zadatke unutar `main` korutine, koristimo *asyncio* Taskove:

> Možemo pozvati korutinu `start_server` unutar `main` korutine

```
async def main():
    asyncio.create_task(start_server())  # Non-blocking pokretanje poslužitelja
    async with aiohttp.ClientSession() as session: # Neka druga asinkrona operacija, npr.
otvaranje klijentske sesije
        print("Klijentska sesija otvorena")
        pass

asyncio.run(main())
```

Primjer: Pokretanje lokalnog poslužitelja i otvaranje klijentske sesije koja šalje zahtjev na taj isti poslužitelj:

```
from aiohttp import web
from aiohttp.web import AppRunner, TCPSite
import asyncio, aiohttp

async def get_users(request):
    return web.json_response({'korisnici': ['Ivo', 'Ana', 'Marko', 'Maja', 'Iva',
'Ivan']})

app = web.Application()
app.router.add_get('/korisnici', get_users)

async def start_server():
    runner = AppRunner(app)
    await runner.setup()
    site = TCPSite(runner, 'localhost', 8080)
    await site.start()
    print("Poslužitelj sluša na http://localhost:8080")

async def main():
    asyncio.create_task(start_server())
    async with aiohttp.ClientSession() as session:
        print("Klijentska sesija otvorena")
        # Ovdje možemo poslati zahtjeve na server
        rezultat = await session.get('http://localhost:8080/korisnici')
        print(await rezultat.text())

asyncio.run(main())
```

Ispisuje:

```
Klijentska sesija otvorena
Poslužitelj sluša na http://localhost:8080
{"korisnici": ["Ivo", "Ana", "Marko", "Maja", "Iva", "Ivan"]}
```

Objašnjenje: kad pokrenemo kôd, prvo će se pokrenuti poslužitelj, a zatim klijentska sesija koja će poslati zahtjev na adresu `http://localhost:8080/korisnici` i ispisati odgovor.

Dobivamo ispis odmah nakon pokretanja skripte:

```
Klijentska sesija otvorena
Poslužitelj sluša na http://localhost:8080
{"korisnici": ["Ivo", "Ana", "Marko", "Maja", "Iva", "Ivan"]}
```

Važno: Nakon završetka `main()` korutine, poslužitelj se gasi. **Pokušaj ponovnog slanja zahtjeva iz terminala ili drugog HTTP klijenta neće uspjeti**.

Idemo vidjeti primjer s **periodičnim pozadinskim zadatkom koji se izvršava istovremeno s poslužiteljem**:

```python
from aiohttp import web
import asyncio


async def get_users(request):
    return web.json_response(
        {"korisnici": ["Ivo", "Ana", "Marko", "Maja", "Iva", "Ivan"]}
    )


# periodična obrada poruka iz reda
async def process_messages():
    while True:
        print("Obrađujem poruke iz reda...")
        await asyncio.sleep(5)


async def start_server():
    app = web.Application()
    app.router.add_get("/korisnici", get_users)

    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, "localhost", 8080)
    await site.start()
    print("Poslužitelj sluša na http://localhost:8080")


async def main():
    asyncio.create_task(process_messages()) # U event loop dodajemo korutinu koja
započinje obradu dolazećih poruka
    asyncio.create_task(start_server()) # Pokrećemo poslužitelj

asyncio.run(main()) # Pokrećemo event loop
```

Pokrenite gore navedeni kôd i u terminalu ćete vidjet ćete ispis korutine `process_messages`, međutim **poslužitelj neće raditi jer se `main()` korutina završava odmah nakon pokretanja pozadinskih zadataka (korutina)**. Obje korutine su beskonačnog trajanja i glavna `main()` korutina se završava odmah nakon njihovoh raspoređivanja u *event loop*.

Da bismo to riješili, moramo naglasiti *event loopu* da ostane aktivan. Ovo je najbolje riješiti korištenjem `await asyncio.Event().wait()` unutar `main()` korutine, što će natjerati event loop da ostane aktivan **dok god se ne dogodi neki vanjski prekid** (npr. `KeyboardInterrupt` signal od korisnika).

Sintaksa:

```
async def main():
    asyncio.create_task(process_messages())
    asyncio.create_task(start_server())
    await asyncio.Event().wait()  # Održava event loop aktivnim dok se ne dogodi neki
oblik prekida, npr. KeyboardInterrupt
```

Rezultat:

```
Obrađujem poruke iz reda...
Poslužitelj sluša na http://localhost:8080
Obrađujem poruke iz reda...
Obrađujem poruke iz reda...
Obrađujem poruke iz reda...
Obrađujem poruke iz reda...
...
```

To je to! Uspjeli smo pokrenuti "dva pozadinska beskonačna zadatka" unutar istog event loopa: HTTP poslužitelj i obradu poruka iz reda. Pokušajte poslati GET zahtjev na `http://localhost:8080/korisnici` kroz neki od HTTP klijenata ili `curl` i vidjet ćete da poslužitelj radi.

```
→ curl http://localhost:8080/korisnici
{"korisnici": ["Ivo", "Ana", "Marko", "Maja", "Iva", "Ivan"]}
```

3.3 HTTP GET ruta s URL (route) parametrima

Nastavljamo nadograđivati naš HTTP poslužitelj.

***Route* parametar** (ili URL parametar) je dinamički dio URL-a koji se koristi za prosljeđivanje podataka unutar same putanje URL-a. Obično se koristi za identifikaciju resursa ili specificiranje dodatnih informacija potrebnih za obradu zahtjeva. Route parametar se ne koristi za filtriranje, sortiranje ili paginaciju podataka (za to se koriste *query* parametri).

Uobičajeno je kada šaljemo HTTP odgovor unutar *handler funkcije*, koristiti `web.json_response()` funkciju te definirati statusni kôd odgovora `status`.

```
async def get_users(request):
  korisnici = [
    {"ime": "Ivo", "godine": 25},
    {"ime": "Ana", "godine": 22},
    {"ime": "Marko", "godine": 19}
  ]
  return web.json_response(korisnici, status=200)
```

GET rutu koja dohvaća točno jednog korisnika, npr. po ID-u, definiramo koristeći HTTP route parametre. U ovom slučaju, parametar rute bi bio `id` korisnika.

Parametre rute iz zahtjeva možemo dohvatiti kroz `request.match_info` rječnik:

```
async def get_users(request):
  user_id = request.match_info['id']

  korisnici = [
    {"id": 1, "ime": "Ivo", "godine": 25},
    {"id": 2, "ime": "Ana", "godine": 22},
    {"id": 3, "ime": "Marko", "godine": 19},
    {"id": 4, "ime": "Maja", "godine": 21},
    {"id": 5, "ime": "Iva", "godine": 40}
  ]

  for korisnik in korisnici:
    if korisnik['id'] == int(user_id):
      return web.json_response(korisnik, status=200)
```

Ako sad pokrenemo kôd dobit ćemo error `500`: `KeyError: 'id'`.

To je zato što nismo definirali:

- *route* parameter `id` u definiciji rute
- slučaj kad korisnik s traženim ID-em ne postoji
- slučaj kad se `id` ne proslijedi u zahtjevu

Dodajemo još jednu definiciju GET rute, ovaj put s *route* parametrom `id`:

```
app.router.add_get('/korisnici/{id}', get_users) # Sada očekujemo route parametar 'id'
```

Možemo upotrijebiti `get()` metodu rječnika kako bismo izbjegli `KeyError`:

> Hint: `get()` metoda vraća `None` ako ključ ne postoji, a možemo definirati i zadani rezultat ako ključ ne postoji

Dakle sljedeći izrazi su ekvivalentni: `request.match_info['id']` → `request.match_info.get('id')`, ali `get()` **metoda je "sigurnija".**

```
async def get_users(request):
  user_id = request.match_info.get('id') # Koristimo get() metodu kako bismo izbjegli
KeyError

  korisnici = [
    {"id": 1, "ime": "Ivo", "godine": 25},
    {"id": 2, "ime": "Ana", "godine": 22},
    {"id": 3, "ime": "Marko", "godine": 19},
    {"id": 4, "ime": "Maja", "godine": 21},
    {"id": 5, "ime": "Iva", "godine": 40}
  ]
```

```
  if user_id is None:
    return web.json_response(korisnici, status=200)

  for korisnik in korisnici:
    if korisnik['id'] == int(user_id):
      return web.json_response(korisnik, status=200)

  return web.json_response({'error': 'Korisnik s traženim ID-em ne postoji'}, status=404)
```

Primjeri slanja HTTP zahtjeva:

GET /korisnici

```
rezultat = await session.get('http://localhost:8080/korisnici')
    rezultat_txt = await rezultat.text()
    print(rezultat_txt)

    rezultat_dict = await rezultat.json() #dekodiraj JSON odgovor u rječnik
    print(rezultat_dict)
```

GET /korisnici/2

```
rezultat = await session.get('http://localhost:8080/korisnici/2')
    rezultat_txt = await rezultat.text()
    print(rezultat_txt)

    rezultat_dict = await rezultat.json() #dekodiraj JSON odgovor u rječnik
    print(rezultat_dict) # {'id': 2, 'ime': 'Ana', 'godine': 22}
```

GET /korisnici/6

```
rezultat = await session.get('http://localhost:8080/korisnici/6')
    rezultat_txt = await rezultat.text()
    print(rezultat_txt)

    rezultat_dict = await rezultat.json() #dekodiraj JSON odgovor u rječnik
    print(rezultat_dict) # {'error': 'Korisnik s traženim ID-em ne postoji'}
```

3.4 Zadaci za vježbu: Interna Klijent-Poslužitelj komunikacija

Zadatak 4: Dohvaćanje proizvoda

Definirajte `aiohttp` poslužitelj koji radi na portu `8081`. Poslužitelj mora imati dvije rute: `/proizvodi` i `/proizvodi/{id}`. Prva ruta vraća listu proizvoda u JSON formatu, a druga rutu vraća točno jedan proizvod prema ID-u. Ako proizvod s traženim ID-em ne postoji, vratite odgovor s statusom `404` i porukom `{'error': 'Proizvod s traženim ID-em ne postoji'}`.

Proizvode pohranite u listu rječnika:

```
proizvodi = [
  {"id": 1, "naziv": "Laptop", "cijena": 5000},
  {"id": 2, "naziv": "Miš", "cijena": 100},
  {"id": 3, "naziv": "Tipkovnica", "cijena": 200},
  {"id": 4, "naziv": "Monitor", "cijena": 1000},
  {"id": 5, "naziv": "Slušalice", "cijena": 50}
]
```

Testirajte poslužitelj na sve slučajeve kroz klijentsku sesiju unutar `main` korutine iste skripte.

Zadatak 5: Proizvodi i ruta za narudžbe

Nadogradite poslužitelj iz prethodnog zadatka na način da podržava i **POST metodu** na putanji `/narudzbe`. Ova ruta prima JSON podatke o novoj narudžbu u sljedećem obliku. Za početak predstavite da je svaka narudžba jednostavna i sadrži samo jedan proizvod i naručenu količinu:

```
{
  "proizvod_id": 1,
  "kolicina": 2
}
```

Handler korutina ove metode mora provjeriti postoji li proizvod s traženim ID-em unutar liste `proizvodi`. Ako ne postoji, vratite odgovor s statusom `404` i porukom `{'error': 'Proizvod s traženim ID-em ne postoji'}`. Ako proizvod postoji, dodajte novu narudžbu u listu narudžbi i vratite odgovor s nadopunjenom listom narudžbi u JSON formatu i prikladnim statusnim kôdom.

Listu narudžbi definirajte globalno, kao praznu listu.

Vaš konačni poslužitelj mora sadržavati 3 rute: `/proizvodi`, `/proizvodi/{id}` i `/narudzbe`.

Testirajte poslužitelj na sve slučajeve kroz klijentsku sesiju unutar `main` korutine iste skripte.

4. WebSocket protokol u `aiohttp` biblioteci

Do sad smo definirali jedan poslužitelj, međutim moguće ih je **unutar jednog procesa definirati** i više. Npr. ako želimo naš mikroservis *exposati* na dva različita protokola: **HTTP** i **WebSocket**, možemo definirati dva različita poslužitelja koji slušaju na različitim portovima i obrađuju zahtjeve.

> WebSocket je protokol koji omogućuje dvosmjernu komunikaciju između klijenta i poslužitelja preko jedne TCP veze, što omogućuje *real-time* interakciju i prijenos podataka bez potrebe za ponovnim uspostavljanjem veze.

WebSocket protokol omogućuje klijentima i poslužiteljima da uspostave trajnu vezu (eng. *persistent connection*) i razmjenjuju podatke u stvarnom vremenu bez potrebe za ponovnim uspostavljanjem veze za svaki zahtjev. Ovo je posebno korisno za aplikacije koje zahtijevaju brzu i kontinuiranu razmjenu podataka, poput chat aplikacija, online igara, financijskih aplikacija i drugih *real-time* sustava.

Unutar `aiohttp` biblioteke, WebSocket podrška je ugrađena i omogućuje jednostavno definiranje WebSocket poslužitelja i klijenata.

Protokol je dostupan unutar modula `aiohttp.web_ws` za poslužitelje i `aiohttp.ClientWebSocketResponse` za klijente.

> Napomena: Ako uključite `web` modul iz `aiohttp` paketa, WebSocket podršku možete koristiti direktno kroz `web.WebSocketResponse` i `web.WSMsgType`, bez potrebe za dodatnim uvozom `web_ws` modula. Ipak, moguće je navesti: `from aiohttp import web_ws` ako želite eksplicitno koristiti samo WebSocket klase i funkcije iz tog modula.

4.1 WebSocket poslužitelj

Primjer mikroservisa s WebSocket **poslužiteljem**:

```
from aiohttp import web
# ili: from aiohttp import web_ws - za eksplcitni uvoz WebSocket modula
import asyncio

async def websocket_handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)

    async for msg in ws:
        if msg.type == web.WSMsgType.TEXT:
            await ws.send_str(f"Primljena poruka: {msg.data}")
        elif msg.type == web.WSMsgType.ERROR:
            print(f'Veza zatvorena s greškom {ws.exception()}')

    print('WebSocket veza zatvorena')
    return ws

app = web.Application()
app.router.add_get('/ws', websocket_handler)
web.run_app(app, host='localhost', port=8080)
```

Ovaj primjer definira WebSocket poslužitelj koji sluša na ruti `/ws`. Kada klijent uspostavi vezu, poslužitelj prima poruke i odgovara s potvrdom primanja.

4.2 WebSocket klijent

Primjer mikroservisa s WebSocket **klijentom**:

```
import aiohttp
import asyncio
```

```
async def websocket_client():
    async with aiohttp.ClientSession() as session:
        async with session.ws_connect('http://localhost:8080/ws') as ws:
            await ws.send_str("Pozdrav, WebSocket poslužitelju!")
            msg = await ws.receive()

            if msg.type == aiohttp.WSMsgType.TEXT:
                print(f"Primljena poruka od poslužitelja: {msg.data}")
            elif msg.type == aiohttp.WSMsgType.ERROR:
                print(f'Veza zatvorena s greškom {ws.exception()}')

asyncio.run(websocket_client())
```

Primjer: Možemo pokrenuti unutar iste skripte koristeći `AppRunner` klasu za pokretanje poslužitelja i istovremeno pokretanje klijentske sesije.

```
import asyncio
from aiohttp import web, ClientSession

# Definicija WebSocket poslužitelja
async def websocket_handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)

    async for msg in ws:
        if msg.type == web.WSMsgType.TEXT:
            await ws.send_str(f"Primljena poruka: {msg.data}")
        elif msg.type == web.WSMsgType.ERROR:
            print(f"Greška na vezi: {ws.exception()}")

    print("WebSocket veza zatvorena")
    return ws

# Pokretanje WebSocket poslužitelja koristeći AppRunner kako bismo mogli paralelno
pokrenuti klijentsku sesiju
async def start_server():
    app = web.Application()
    app.router.add_get("/ws", websocket_handler)

    runner = web.AppRunner(app)
    await runner.setup()

    site = web.TCPSite(runner, "localhost", 8080)
    await site.start()

    print("Poslužitelj pokrenut na http://localhost:8080")
    return runner

# Definicija WebSocket klijenta
async def run_client():
    async with ClientSession() as session:
```

```
        async with session.ws_connect("http://localhost:8080/ws") as ws:
            await ws.send_str("Pozdrav, WebSocket poslužitelju!")
            msg = await ws.receive()

            if msg.type == web.WSMsgType.TEXT:
                print(f"Klijent primio: {msg.data}")
            else:
                print(f"Neočekivana poruka: {msg}")



async def main():
    runner = await start_server()

    # pričekaj da se poslužitelj stabilizira
    await asyncio.sleep(0.2)

    await run_client()
    await runner.cleanup()

asyncio.run(main())
```

Rezultat:

```
Poslužitelj pokrenut na http://localhost:8080
Klijent primio: Primljena poruka: Pozdrav, WebSocket poslužitelju!
WebSocket veza zatvorena
```

5. Podjela kôda u više datoteka (1 servis = 1 datoteka)

Naučili smo kako definirati `aiohttp` poslužitelje i klijentske sesije, kako definirati rute i *handler* funkcije, kako slati HTTP zahtjeve i obrađivati odgovore. Međutim, sve smo to radili unutar jedne skripte - `index.py`.

Vidjeli smo da Python omogućuje pokretanje poslužitelja i paralelno stvaranje klijentskih sesija za slanje zahtjeva unutar iste skripte koristeći `AppRunner` klasu.

Ono što je ključno - do sad se sve izvršavalo u jednom threadu, odnosno **unutar jednog procesa**. Međutim, kad pričamo o mikroservisnoj arhitekturi, **pričamo o više poslužitelja i više klijenata koji komuniciraju međusobno**.

Naš sljedeći *challenge* je - **podijeliti kôd u više datoteka**, odnosno definirati poslužitelje i klijentske sesije u zasebnim skriptama.

5.1 Jednostavna simulacija mikroservisne arhitekture

Neka nam trenutna asocijacija za mikroservis bude **web poslužitelj**, odnosno nekakav REST API klijent koji sluša na određenoj adresi i portu te obrađuje dolazne zahtjeve. U našem slučaju, to će biti `aiohttp` poslužitelj. S druge strane, **klijent** će biti `aiohttp` klijentska sesija koja šalje zahtjeve prema poslužitelju.

Izradimo novi direktorij `microservice_simulation`.

U direktoriju `microservice_simulation` izradite sljedeće datoteke:

- `client.py` - ovdje ćemo definirati klijentsku sesiju
- `microservice_1.py` - ovdje ćemo definirati prvi mikroservis (i njegov poslužitelj)
- `microservice_2.py` - ovdje ćemo definirati drugi mikroservis (i njegov poslužitelj)

```
→ mkdir microservice_simulation
→ cd microservice_simulation

→ touch client.py, microservice_1.py, microservice_2.py
```

Krenimo s definicijom poslužitelja u `microservice_1.py` datoteci. Svaki servis će imati jednostavnu korijensku rutu `/` koja vraća poruku `"Hello from Microservice X"`.

`microservice_1` neka sluša na portu `8081`:

```
# microservice_1.py
from aiohttp import web

async def handle_service1(request):
  return web.json_response({"message": "Hello from Microservice 1"})

app = web.Application()
app.router.add_get('/', handle_service1)

web.run_app(app, port=8081)
```

U `microservice_2.py` datoteci ćemo definirati drugi mikroservis koji sluša na portu `8082`:

```
# microservice_2.py
from aiohttp import web

async def handle_service2(request):
  return web.json_response({"message": "Hello from Microservice 2"})

app = web.Application()
app.router.add_get('/', handle_service2)

web.run_app(app, port=8082)
```

U klijentskoj sesiji tj. `client.py` datoteci ćemo prvo definirati glavnu korutinu `main`.

```
# client.py
import aiohttp
import asyncio

async def main():
  print("Pokrećem main korutinu")
  pass

asyncio.run(main())
```

Što dalje? **Velika greška** bila bi uključiti ove dvije datoteke unutar `client.py` datoteke koristeći `import` naredbu.

```
# client.py
import aiohttp
import asyncio
import microservice_1 # ? NIKAKO
import microservice_2 # ? NIKAKO

async def main():
  print("Pokrećem main korutinu")
  pass

asyncio.run(main())
```

Ako pokrenete `client.py`, vidjet ćete sljedeći ispis u terminalu:

```
======== Running on http://0.0.0.0:8081 ========
(Press CTRL+C to quit)
^C
======== Running on http://0.0.0.0:8082 ========
(Press CTRL+C to quit)
^C
Pokrećem main korutinu
```

Na ovaj način, jednostavno smo "kopirali" kôd iz ova dva poslužitelja i zaljepili ga na početak `client.py` datoteke. Pokretanjem skripte vidimo da se oba poslužitelja pokreću, ali tek nakon što ih ugasimo pokreće se `main` korutina u `client.py`.

Već smo rekli mikroservisnu arhitekturu ne želimo zamišljati kao "jedan veliki monolitni kôd", odnosno veliki program koji putem vanjskih biblioteka/modula dobiva na složenosti/raspodijeljenosti, već **želimo pokrenuti više manjih i jednostavnijih servisa koji međusobno komuniciraju preko mreže** (kroz HTTP/WS ili neki treći komunikacijski protokol).

5.1.1 Pokretanje više mikroservisa

Potrebno je pokrenuti poslužitelje mikroservisa samostalno iz terminala, a zatim pokrenuti klijentsku sesiju iz `client.py` datoteke. Međutim, do sad ste vidjeli da kad pokrenemo jedan poslužitelj, on blokira izvođenje ostatka kôda. **Rješenje je sljedeće**: pokrenuti svaki poslužitelj u zasebnom procesu, a to je najlakše postići **pokretanjem više terminal sesija**.

Skriptu pokrećemo naredbom `python microservice_1.py` u jednom terminalu, a drugu skriptu u drugom terminalu.

Prisjetite se varijable `__name__` koja sadrži naziv trenutačnog modula Definirali smo uvjetu izjavu `if __name__ == '__main__':` kako bismo osigurali da se kôd unutar bloka izvršava samo ako je skripta pokrenuta direktno, a ne uvezena kao modul. **Upravo to nam i treba.**

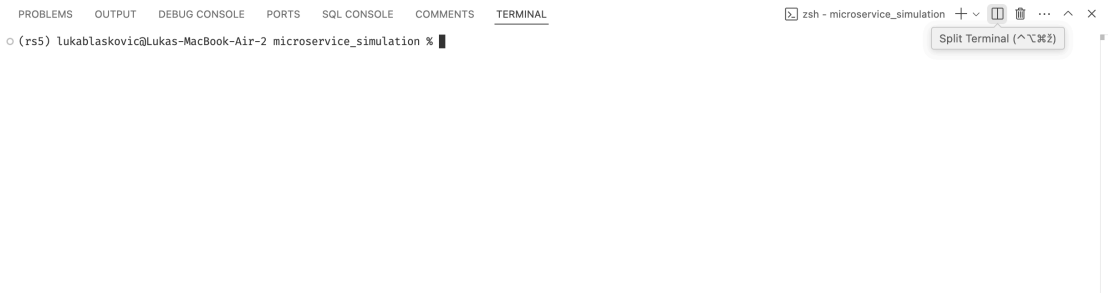
Pokretanje poslužitelja u svakom mikroservisu ćemo omotati u `if __name__ == '__main__':` uvjetnu izjavu:

```
# microservice_1.py
from aiohttp import web

async def handle_service1(request):
  return web.json_response({"message": "Hello from Microservice 1"})

app = web.Application()
app.router.add_get('/', handle_service1)

if __name__ == "__main__":
  web.run_app(app, port=8081)
```

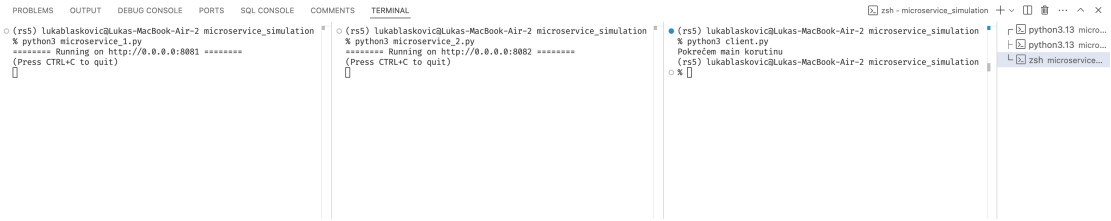
I drugi:

```
# microservice_2.py
from aiohttp import web

async def handle_service2(request):
    return web.json_response({"message": "Hello from Microservice 2"})

app = web.Application()
app.router.add_get('/', handle_service2)

if __name__ == "__main__":
    web.run_app(app, port=8082)
```

Ako koristite VS Code, terminale možete jednostavno podijeliti koristeći opciju `Split Terminal` (`Ctrl/CMD` + `Shift` + `5`).



Podijelite terminal na tri dijela, jedan za svaki mikroservis i jedan za klijenta.

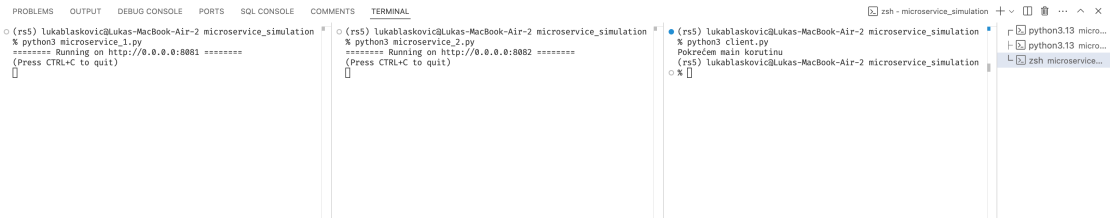


Pokrenite svaki mikroservis u zasebnom terminalu:

```
python3 microservice_1.py # Terminal 1
python3 microservice_2.py # Terminal 2
```

Možete pokrenuti i klijenta:

```
python3 client.py # Terminal 3
```



Na ovaj način, sve smo podijelili u **zasebne datoteke**, a samim tim i **zasebne procese**. Sada ćemo mikroservise pustiti na miru te implementirati slanje zahtjeva iz `client.py`.

Možemo definirati dvije korutine, jednu za svaki mikroservis, unutar `client.py` datoteke.

U svakoj korutini ćemo otvoriti klijentsku sesiju i poslati zahtjev na odgovarajući mikroservis i njegov endpoint.

```
# client.py

async def fetch_service1():
  async with aiohttp.ClientSession() as session:
    response = await session.get('http://localhost:8081/')
    return await response.json()

async def fetch_service2():
  async with aiohttp.ClientSession() as session:
    response = await session.get('http://localhost:8082/')
    return await response.json()
```

Možemo poslati zahtjeve sekvencijalno unutar `main` korutine:

```
# client.py

async def main():
  print("Pokrećem main korutinu")
  service1_response = await fetch_service1()
  print(f"Odgovor mikroservisa 1: {service1_response}")

  service2_response = await fetch_service2()
  print(f"Odgovor mikroservisa 2: {service2_response}")
```

Pokrenite kôd, trebali biste dobiti ispis:

```
Pokrećem main korutinu
Odgovor mikroservisa 1: {'message': 'Hello from Microservice 1'}
Odgovor mikroservisa 2: {'message': 'Hello from Microservice 2'}
```

5.1.2 Konkurentno slanje zahtjeva

Kako zahtjeve poslati konkurentno? Još jednostavnije!

```
# client.py

async def main():
  print("Pokrećem main korutinu")
  results = await asyncio.gather(fetch_service1(), fetch_service2()) # konkurentno slanje
zahtjeva, vraća listu rječnika
  print(results)
```

ili

```
# client.py

async def main():
  print("Pokrećem main korutinu")
  service1_response, service2_response = await asyncio.gather( # konkurentno slanje
zahtjeva, vraća tuple rječnika
      fetch_service1(),
      fetch_service2()
  )
  print(service1_response, service2_response)
```

Česta greška kod konkurentnog slanja: Recimo da želimo napisati samo jednu korutinu `fetch_service()` koja će slati zahtjeve na oba mikroservisa. Tada bi unutar te korutine slali 2 zahtjeva, bilo **kroz jednu ili dvije klijentske sesije**.

Primjer slanja zahtjeva otvaranjem dvije klijentske sesije:

```
async def fetch_service():
  async with aiohttp.ClientSession() as session:
    # Klijentska sesija za mikroservis 1
    async with session.get('http://localhost:8081/') as response1:
        service1_data = await response1.json()
    # Klijentska sesija za mikroservis 2
    async with session.get('http://localhost:8082/') as response2:
        service2_data = await response2.json()

  return service1_data, service2_data
```

U `main` korutini jednostavno pozivamo ovu korutinu:

```
async def main():
  print("Pokrećem main korutinu")
  service1_response, service2_response = await fetch_service() # kôd nije konkurentan, ali je napisan asinkrono.
  print(service1_response, service2_response)
```

Ovaj kôd nije konkurentan jer se zahtjevi u korutini `fetch_service` šalju i čekaju sekvencijalno, a ne konkurentno.

Što ako dodamo `gather` u main korutinu?

```
async def main():
  print("Pokrećem main korutinu")
  results = await asyncio.gather(fetch_service()) # je li kôd sada konkurentan?
  print(results)
```

Je li kôd sada konkurentan?

▸ Spoiler alert! Odgovor na pitanje

Međutim, zašto ne bi mogli koristiti `gather` u `fetch_service()` korutini?

Ideja je sljedeća: **idemo otvoriti jednu klijentsku sesiju i unutar nje slati zahtjeve na oba mikroservisa**, budući da možemo definirati različiti URL za svaki `session.get()` poziv.

```
# client.py

async def fetch_service():
  async with aiohttp.ClientSession() as session:
    service_1 = await session.get('http://localhost:8081/')
    service_2 = await session.get('http://localhost:8082/')

    rezultati = await asyncio.gather(
      service_1,
      service_2
    )

    return rezultati
```

Postoji problem u kôdu iznad. Možete li ga pronaći?

▸ Spoiler alert! Odgovor na pitanje

kôd daje sljedeću grešku:

```
   rezultati = await asyncio.gather(
                     ~~~~~~~~~~~~~~^
      service_1,
      ^^^^^^^^^^
      service_2
      ^^^^^^^^^
    )
    ^
  ...
    raise TypeError('An asyncio.Future, a coroutine or an awaitable '
                    'is required')
TypeError: An asyncio.Future, a coroutine or an awaitable is required
[nodemon] app crashed - waiting for file changes before starting...
```

Kako pročitati grešku? **TypeError: An asyncio.Future, a coroutine or an awaitable is required** (proslijedili smo krivi input u `gather` funkciju, mora biti korutina ili `awaitable` objekt korutine ili Task)

Rješenje je jednostavno: `service_1` i `service_2` su objekti tipa `ClientResponse`, a ne korutine (zato što smo ih već *awaitali*, tj. korutine su se izvršile). Ako odradimo deserijalizaciju odgovora, možemo vidjeti da su to rječnici.

```
print(type(await service_1.json()), type(await service_1.json())) # <class 'dict'> <class
'dict'>
```

Prisjetite se kako riješiti ovaj problem? *Kada želimo neku korutinu pohraniti za kasnije izvršavanje, što koristimo...?*

▸ Spoiler alert! Odgovor na pitanje

```
# client.py
```

```
async def fetch_service():
  async with aiohttp.ClientSession() as session:
    service_1 = session.get('http://localhost:8081/')
    service_2 = session.get('http://localhost:8082/')

    tasks = [asyncio.create_task(service_1), asyncio.create_task(service_2)]
    rezultati = await asyncio.gather(*tasks)

    return rezultati

async def main():
  print("Pokrećem main korutinu")
  results = await fetch_service()
  print(results)

asyncio.run(main())
```

Pokrenite kôd, vidjet ćete ispis:

```
Pokrećem main korutinu
[<ClientResponse(http://localhost:8081/) [200 OK]>
<CIMultiDictProxy('Content-Type': 'application/json; charset=utf-8', 'Content-Length':
'40', 'Date': 'Wed, 04 Dec 2024 00:49:08 GMT', 'Server': 'Python/3.13 aiohttp/3.11.7')>
, <ClientResponse(http://localhost:8082/) [200 OK]>
<CIMultiDictProxy('Content-Type': 'application/json; charset=utf-8', 'Content-Length':
'40', 'Date': 'Wed, 04 Dec 2024 00:49:08 GMT', 'Server': 'Python/3.13 aiohttp/3.11.7')>
]
```

Radi! Ali odgovori su tipa `ClientResponse`. Još moramo odraditi deserijalizaciju.

Možemo ju jednostavno direktno odraditi na izlasku iz funkcije.

Imamo listu `ClientResponse` rezultata, a želimo listu raspakiranih podataka (rječnika). Metoda za deserijalizaciju je `response.json()`, a sve možemo definirati u jednoj liniji koristeći **list comprehension** i/ili **map funkciju?**

```
# client.py

async def fetch_service():
  async with aiohttp.ClientSession() as session:
    service_1 = session.get('http://localhost:8081/')
    service_2 = session.get('http://localhost:8082/')

    tasks = [asyncio.create_task(service_1), asyncio.create_task(service_2)]
    rezultati = await asyncio.gather(*tasks)

    return [await rezultat.json() for rezultat in rezultati] # radi!
```

ili:

```
return list(map(lambda rezultat: await rezultat.json(), rezultati)) # ili ne možemo ? :)
```

Ako pokrenete korutinu s drugom `return` dobit ćete grešku: `SyntaxError: 'await' outside function`, iako ga koristimo unutar korutine `fetch_service()`. Zašto?

Problem je što `await` ustvari koristimo unutar funkcije `map` koja nije korutina, niti je funkcija namijenjena za asinkrono izvršavanje. `lambda` koju prosljeđujemo `map` funkciji nije korutina već je sinkrona funkcija. **Zato je bolje koristiti list comprehension**.

Kako možemo dokazati da je ovaj kôd uistinu konkurentan? Simulacijom čekanja (`asyncio.sleep` i mjerenjm vremena: `time` modul).

> Pokušajte prvo sami, a zatim provjerite rješenje u nastavku.

Rješenje:

```
# microservice_1.py
from aiohttp import web
from asyncio import sleep
async def handle_service1(request):
  await sleep(1)
  return web.json_response({"message": "Hello from Microservice 1"})

app = web.Application()
app.router.add_get('/', handle_service1)

if __name__ == "__main__":
    web.run_app(app, port=8081)
```

```
# microservice_2.py
from aiohttp import web
from asyncio import sleep

async def handle_service2(request):
  await sleep(2)
  return web.json_response({"message": "Hello from Microservice 2"})

app = web.Application()
app.router.add_get('/', handle_service2)

if __name__ == "__main__":
    web.run_app(app, port=8082)
```

```
# client.py
import aiohttp
import asyncio
import time

async def fetch_service():
```

```
    async with aiohttp.ClientSession() as session:
        service_1 = session.get('http://localhost:8081/')
        service_2 = session.get('http://localhost:8082/')

        tasks = [asyncio.create_task(service_1), asyncio.create_task(service_2)]
        rezultati = await asyncio.gather(*tasks)

        return [await rezultat.json() for rezultat in rezultati] # radi!

async def main():
    print("Pokrećem main korutinu")
    start_time = time.time()
    results = await fetch_service()
    end_time = time.time()
    print(results)
    print(f"Vrijeme izvršavanja: {end_time - start_time:.2f} sekundi")

asyncio.run(main())
```

Ako pokrenete kôd vidjet ćete da je vrijeme izvršavanja `~2 sekunde`, a ne `~3 sekunde` kako bi bilo da se zahtjevi šalju sekvencijalno.

5.2 Simulacija mikroservisne arhitekture: Računske operacije

U prethodnom primjeru, simulirali smo mikroservisnu arhitekturu kroz dva jednostavna mikroservisa koji su vraćali poruke. U stvarnosti, mikroservisi obavljaju različite zadatke, od jednostavnih do složenih. Sada ćemo pokušati definirati nešto "zanimljivije": mikroservise koji obavljaju računske operacije.

Ovu arhitekturu definirat ćemo unutar direktorija `microservice_calculations`.

5.2.1 Sekvencijalna obrada podataka

Ideja je sljedeća:

- definirat **ćemo dva mikroservisa i njihove HTTP poslužitelje** koji obavljaju računske operacije
- definirat **ćemo jednog HTTP klijenta** koji šalje zahtjeve u obliku lista brojeva

1. mikroservis će računati zbroj svih brojeva i vratiti rezultat
2. mikroservis će upotrijebiti rezultat prvog mikroservisa i izračunati omjer svakog broja s ukupnim zbrojem

Prvo ćemo definirati klijenta:

```
→ mkdir microservice_calculations
→ cd microservice_calculations
→ touch client.py
```

U `client.py` datoteci definirajmo `main` korutinu.

```
# client.py

import aiohttp
import asyncio

async def main():
  print("Pokrećem main korutinu")
  pass

asyncio.run(main())
```

Idemo definirati prvi mikroservis koji će računati zbroj svih brojeva.

```
→ touch microservice_sum.py
```

```
# microservice_sum.py

from aiohttp import web
# koji endpoint moramo definirati?
app = web.Application()

web.run_app(app, host='localhost', port=8081)
```

Kako servis očekuje ulazne podatke, moramo definirati `PORT` rutu i odgovarajuću *handler* korutinu:

```
# microservice_sum.py
from aiohttp import web

async def handle_zbroj(request):
  data = await request.json()
  zbroj = sum(data)
  return web.json_response({"zbroj": zbroj})

app = web.Application()
app.router.add_post('/zbroj', handle_zbroj)
web.run_app(app, host='localhost', port=8081)
```

Testirat ćemo prvo ovaj mikroservis kroz HTTP klijent. Kako poslati podatke?

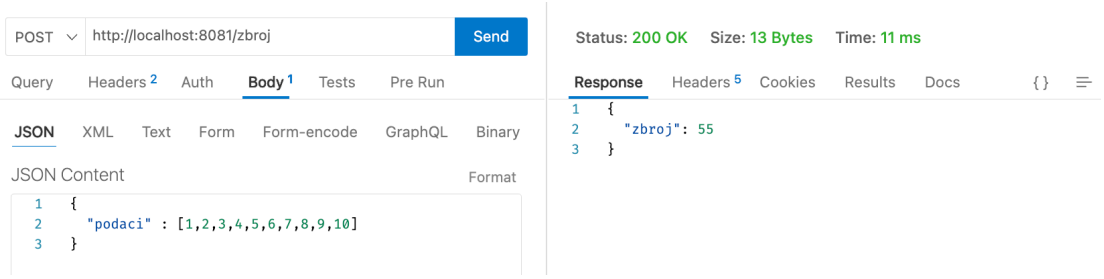
HTTP zahtjeve želimo pisati u JSON formatu, a **uobičajeno je da JSON format sadrži uvijek barem 1 ključ**.

Definirat ćemu listu u ključu `'podaci'`:

```
{
  "podaci" : [1,2,3,4,5,6,7,8,9,10]
}
```

Kako bismo sada ispravno obradili ovaj zahtjev, moramo nakon deserijalizacije dohvatiti listu podataka iz ključa `'podaci'`.

```
# microservice_sum.py

async def handle_zbroj(request):
  data = await request.json()
  data_brojevi = data.get("podaci") # ili data['podaci']
  zbroj = sum(data_brojevi)
  return web.json_response({"zbroj": zbroj})
```



U HTTP klijentu radi. Još moramo stvari prebaciti u `client.py`:

```
# client.py

async def main():
  print("Pokrećem main korutinu")
  data = [i for i in range (1, 11)]
  data_json = {"podaci": data} # JSON format (dodajemo ključ 'podaci')
  async with aiohttp.ClientSession() as session:
    response = await session.post('http://localhost:8081/zbroj', json=data_json)
    print(await response.json())

asyncio.run(main())
```

Pokrenite mikroservis i klijenta. Trebali biste dobiti ispis:

```
Pokrećem main korutinu
{'zbroj': 55}
```

Sada ćemo definirati drugi mikroservis koji će koristiti rezultat prvog mikroservisa i izračunati omjer svakog broja s ukupnim zbrojem.

```
→ touch microservice_ratio.py
```

Stvari su vrlo slične, samo naš POST endpoint sad zaprima 2 ključa: `'podaci'` i `'zbroj'`.

```
# microservice_ratio.py

import aiohttp
from aiohttp import web
import asyncio

app = web.Application()
```

```
async def handle_ratio(request):
  data = await request.json()
  data_brojevi = data.get("podaci")
  data_zbroj = data.get("zbroj")
  ratio_list = [i / data_zbroj for i in data_brojevi] # vraćamo listu omjera za svaki broj
  return web.json_response({"ratio_list": ratio_list})

app.router.add_post('/ratio', handle_ratio)

web.run_app(app, host='localhost', port=8082)
```

Dakle, mikroservis na ruti `/ratio` očekuje tijelo HTTP zahtjeva u obliku:

```
{
  "podaci": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
  "zbroj": 55
}
```

- gdje `'podaci'` predstavlja listu brojeva
- a `'zbroj'` je rezultat mikroservisa `microservice_sum`

Prvo ćemo poslati zahtjev na prvi mikroservis, zatim rezultat ovog zahtjeva koristiti kao input za drugi mikroservis.

```
# client.py
async def main():
  print("Pokrećem main korutinu")
  data = [i for i in range (1, 11)]
  data_json = {"podaci": data}
  async with aiohttp.ClientSession() as session:
    # slanje zahtjeva na 1. mikroservis
    microservice_sum_result = await session.post('http://localhost:8081/zbroj',
json=data_json)
    microservice_sum_data = await microservice_sum_result.json() # podaci iz odgovora 1.
mikroservisa
    zbroj = microservice_sum_data.get("zbroj")

    # slanje zahtjeva na 2. mikroservis
    microservice_ratio_result = await session.post('http://localhost:8082/ratio', json=
{"podaci": data, "zbroj": zbroj})
    microservice_ratio_data = await microservice_ratio_result.json() # podaci iz odgovora
2. mikroservisa
    ratio_list = microservice_ratio_data.get("ratio_list")

    print(f"Zbroj: {zbroj}")
    print(f"Lista omjera: {ratio_list}")

asyncio.run(main())
```

Pokrenite sve mikroservise i klijenta. Trebali biste dobiti sljedeći ispis:

```
Pokrećem main korutinu
Zbroj: 55
Lista omjera: [0.01818181818181818, 0.03636363636363636, 0.05454545454545454,
0.07272727272727272, 0.09090909090909091, 0.10909090909090909, 0.12727272727272726,
0.14545454545454545, 0.16363636363636364, 0.18181818181818182]
```

Još ćemo samo zaokružiti omjere na dvije decimale.

```
ratio_list = [round(i / data_zbroj, 2) for i in data_brojevi]
```

Provjerite ispis:

```
Pokrećem main korutinu
Zbroj: 55
Lista omjera: [0.02, 0.04, 0.05, 0.07, 0.09, 0.11, 0.13, 0.15, 0.16, 0.18]
```

5.2.2 Konkurentna obrada podataka (osnovno)

U prethodnom primjeru, zahtjevi su se **slali sekvencijalno i bili obrađeni sekvencijalno**.

Razlog tomu je što svakako moramo dobiti rezultat izvođenja prvog mikroservisa prije nego što pošaljemo zahtjev na drugi mikroservis, budući da nam treba rezultat prvog mikroservisa kao ulaz za drugi mikroservis.

Bez obzira što je taj rezultat u ovom slučaju vrlo banalan (običan zbroj brojeva u listi) **u stvarnosti se radi o puno složenijim operacijama**.

Glavni izazov u konkurentnom izvršavanju slanja zahtjeva koji smo do sada uočili je upravo ova **nekonzistentnost u obradi podataka**. Zamislite da, zbog performansi sustava, želimo poslati 10 000 zahtjeva kroz 10 različitih mikroservisa (npr. kako bismo ubrzali obradu rezultata za onih ~80%), od kojih neki ovise o rezultatima drugih. U tom slučaju, konkurentno slanje zahtjeva koje smo dosad radili nije dovoljno, jer se zahtjevi šalju i čekaju nasumično (puno parametra je van naše kontrole, npr. propusnost mreže, latencija, opterećenje poslužitelja itd.).

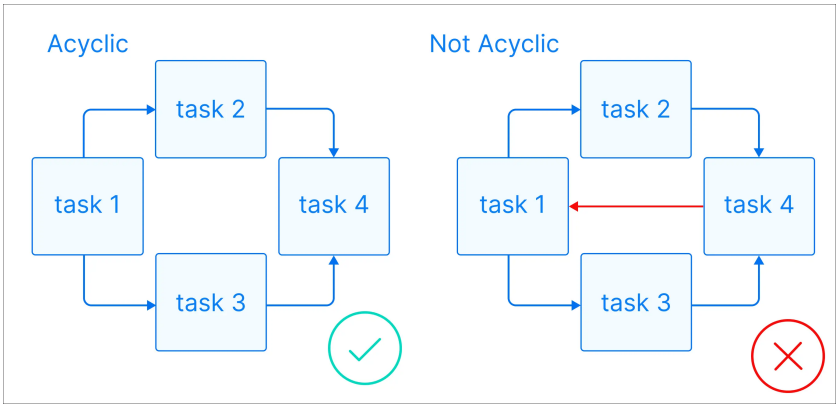
Primjerice, definiramo listu od 10 taskova:

```
tasks = [task1, task2, task3, task4, task5, task6, task7, task8, task9, task10]

# Ali postoje ovisnosti među taskovima:
# Npr. Taskovi 5-10 ovise o taskovima 1-4

rezultati = await asyncio.gather(*tasks) # konkurentno slanje zahtjeva
```

Što nas muči? Recimo da taskovi 5-10 ovise o rezultatima taskova 1-4. Kako osigurati da se taskovi 5-10 izvrše tek nakon što se izvrše taskovi 1-4? Odnosno, bolje pitanje bi glasilo: **Kako upravljati konkurentnom egzekucijom međusobno ovisnih taskova?**

Skupina taskova koji se mogu izvršiti paralelno bez ovisnosti naziva se **batch**. U našem slučaju, taskovi 1-4 mogu se izvršiti paralelno, a tek nakon što se svi ti taskovi završe, možemo krenuti s izvršavanjem taskova 5-10. Ipak, na ovaj način gubimo na performansama jer čekamo da se svi taskovi iz batcha 1 završe prije nego krenemo s batchom 2. Također, ovisnosti između taskova mogu biti složenije prirode (npr. task 6 ovisi o tasku 2, task 7 ovisi o tasku 3, task 2 ovisi o tasku 9 itd.). Upravljanje ovakvim ovisnostima može postati vrlo složeno.

DAG (*eng. Directed Acyclic Graph*) je matematička struktura koja se često koristi za **modeliranje ovisnosti između taskova**. U DAG-u, **čvorovi** predstavljaju Taskove, a usmjereni **bridovi** predstavljaju ovisnosti između njih. Na ovaj način možemo jasno vidjeti koje taskove možemo izvršiti paralelno i koje taskove moramo čekati da se završe prije nego krenemo s određenim taskom. Aciklični graf nema ciklusa (što znači da ne možemo imati situaciju gdje task A ovisi o tasku B, a task B ovisi o tasku A).



> Directed Acyclic Graph (DAG) primjer s ovisnostima između taskova, izvor: https://www.astronomer.io/docs/learn/dags

U stvarnom svijetu, upravljanje ovakvim ovisnostima može biti vrlo složeno. Postoje specijalizirani alati i biblioteke (npr. Apache Airflow, Luigi, Prefect) koji pomažu u upravljanju ovakvim DAG-ovima i izvršavanju taskova na temelju njihovih ovisnosti.

Ovoga ćemo se dotaknuti na budućim vježbama, a za sada ćemo izmijeniti naš kod kako bi mikroservisi (Taskovi) bili nezavisni jedan o drugom.

- Neka prvi mikroservis vraća kvadrate brojeva, a drugi mikroservis vraća njihove kvadratne korijene.

Sada imamo **isti resurs za oba mikroservisa**, a to su brojevi. Kao rezultat na klijentskoj strani želimo zbrojiti **zbroj kvadrata** i **zbroj kvadratnih korijena**.

Definiramo `microservice_square.py`:

```
touch microservice_square.py
```

```
# microservice_square.py
from aiohttp import web

async def handle_squares(request):
  data = await request.json()
  data_brojevi = data.get("podaci")
  kvadrati = [i ** 2 for i in data_brojevi]
  return web.json_response({"kvadrati": kvadrati})

app = web.Application()
app.router.add_post('/kvadrati', handle_squares)
web.run_app(app, host='localhost', port=8083)
```

Mikroservis `microservice_sqrt.py` koji računa i vraća korijene brojeva:

```
→ touch microservice_sqrt.py
```

```
# microservice_sqrt.py
from aiohttp import web

async def handle_squares(request):
  data = await request.json()
  data_brojevi = data.get("podaci")
  korijeni = [i ** 0.5 for i in data_brojevi]
  return web.json_response({"korijeni": korijeni})

app = web.Application()
app.router.add_post('/korijeni', handle_squares)
web.run_app(app, host='localhost', port=8084)
```

Pokrenite ove mikroservise.

Zahtjeve možemo obraditi konkurentno koristeći `gather` funkciju:

```
# client.py

import aiohttp
import asyncio


async def fetch_square_data(session, data_json):
  response = await session.post('http://localhost:8083/kvadrati', json=data_json)
  return await response.json()

async def fetch_sqrt_data(session, data_json):
  response = await session.post('http://localhost:8084/korijeni', json=data_json)
  return await response.json()

async def main():
```

```
    print("Pokrećem main korutinu")
    data = [i for i in range(1, 11)]
    data_json = {"podaci": data} # resurs je isti za oba mikroservisa

    async with aiohttp.ClientSession() as session:
        # Konkurentno pozivanje mikroservisa
        microservice_square_data, microservice_sqrt_data = await
asyncio.gather(fetch_square_data(session, data_json), fetch_sqrt_data(session, data_json))

        # Ekstrakcija podataka
        kvadrati = microservice_square_data.get("kvadrati")
        korijeni = microservice_sqrt_data.get("korijeni")

        print(f"Zbroj kvadrata: {sum(kvadrati)}")
        print(f"Zbroj korijena: {sum(korijeni)}")
        print(f"Ukupni zbroj: {sum(kvadrati) + sum(korijeni)}")

asyncio.run(main())
```

Testirajte kôd:

```
Pokrećem main korutinu
Zbroj kvadrata: 385
Zbroj korijena: 22.4682781862041
Ukupni zbroj: 407.4682781862041
```

6. Zadaci za vježbu: Mikroservisna arhitektura - razvoj aiohttp poslužitelja i klijenata

Zadatak 6: Jednostavna komunikacija

Definirajte 2 mikroservisa u 2 različite datoteke. Prvi mikroservis neka sluša na portu `8081` i na endpointu `/pozdrav` vraća JSON odgovor nakon 3 sekunde čekanja, u formatu: `{"message": "Pozdrav nakon 3 sekunde"}`. Drugi mikroservis neka sluša na portu `8082` te na istom endpointu vraća JSON odgovor nakon 4 sekunde: `{"message": "Pozdrav nakon 4 sekunde"}`.

Unutar `client.py` datoteke definirajte 1 korutinu koja može slati zahtjev na oba mikroservisa, mora primati argumente `url` i `port`. Korutina neka vraća JSON odgovor.

Korutinu pozovite unutar `main` korutine. **Prvo demonstrirajte sekvencijalno slanje zahtjeva, a zatim konkurentno slanje zahtjeva.**

Zadatak 7: Računske operacije

Definirajte 3 mikroservisa unutar direktorija `microservice_calculations`. Prvi mikroservis neka sluša na portu `8083` i na endpointu `/zbroj` vraća JSON bez čekanja. Ulazni podatak u tijelu zahtjeva neka bude lista brojeva, a odgovor neka bude zbroj svih brojeva. Dodajte provjeru ako brojevi nisu proslijeđeni, vratite odgovarajući HTTP odgovor i statusni kôd.

Drugi mikroservis neka sluša na portu `8084` te kao ulazni podataka prima iste podatke. Na endpointu `/umnozak` neka vraća JSON odgovor s umnoškom svih brojeva. Dodajte provjeru ako brojevi nisu proslijeđeni, vratite odgovarajući HTTP odgovor i statusni kôd.

Treći mikroservis pozovite nakon konkurentnog izvršavanja prvog i drugog mikroservisa. Dakle treći ide sekvencijalno jer mora čekati rezultati prethodna 2. Ovaj mikroservis neka sluša na portu `8085` te na endpointu `/kolicnik` očekuje JSON s podacima prva dva servisa. Kao odgovor mora vratiti količnik umnoška i zbroja. Dodajte provjeru i vratite odgovarajući statusni kôd ako se pokuša umnožak dijeliti s 0.

U `client.py` pozovite konkurentno s proizvoljnim podacima prva dva mikroservisa, a zatim sekvencijalno pozovite treći mikroservis.

Zadatak 8: Mikroservisna obrada - CatFacts API

Definirajte 2 mikroservisa unutar direktorija `cats`.

Prvi mikroservis `cat_microservice.py` mora slušati na portu `8086` i na endpointu `/cats` vraćati JSON odgovor s listom činjenica o mačkama. Endpoint `/cat` mora primati URL parametar `amount` koji predstavlja broj činjenica koji će se dohvatiti. Na primjer, slanjem zahtjeva na `/cat/30` dohvatit će se 30 činjenica o mačkama. Činjenice se moraju dohvaćati **konkurentnim slanjem zahtjeva na CatFacts API**. Link: https://catfact.ninja/

Drugi mikroservis `cat_fact_check` mora slušati na portu `8087` i na endopintu `/facts` očekivati JSON objekt s listom činjenica o mačkama u tijelu HTTP zahtjeva. Glavna dužnost ovog mikroservisa je da provjeri svaku činjenicu sadrži li riječ `cat` ili `cats`, neovisno o velikim i malim slovima. Odgovor neka bude JSON objekt s novom listom činjenica koje zadovoljavaju prethodni uvjet.

U `client.py` pozovite ove dvije korutine sekvencijalno, obzirom da drugi mikroservis ovisi o rezultatima prvog. Testirajte kôd za proizvoljan broj činjenica.