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 suvremeni pristup dizajnu softvera, gdje se aplikacija gradi kao skup manjih, samostalnih servisa koji međusobno komuniciraju putem mreže. Svaki od tih servisa može se promatrati kao zasebna jedinica koja obavlja jednu specifičnu funkciju, pri čemu komunikacija najčešće koristi standardizirane protokole poput HTTP-a. Za razliku od tradicionalne monolitne arhitekture, u kojoj su svi dijelovi aplikacije objedinjeni u jednoj cjelini, mikroservisna arhitektura razdvaja ključne funkcionalnosti poput poslovne logike, baza podataka, autentifikacije i drugih komponenti u odvojene servise.

Ovakav pristup donosi brojne prednosti: omogućuje veću skalabilnost i pouzdanost sustava, olakšava organizaciju velikih razvojnih timova te ubrzava proces implementacije i unapređenja rješenja. Mikroservisi tako postaju temelj fleksibilnog, održivog i modernog razvoja softvera.

🆙 **Posljednje ažurirano: 27.11.2024.**

- skripta nije dovršena.

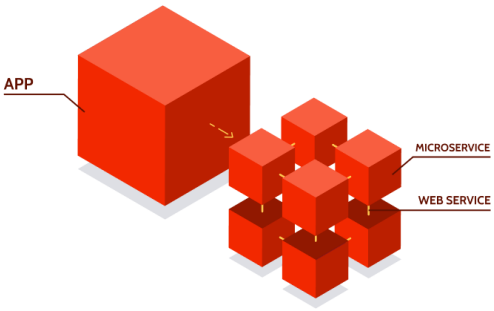
Sadržaj

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 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 i modularnost sustava.

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

- U mikroservisnoj arhitekturi, servisi se obično implementiraju kao **odvojeni procesi** koji međusobno komuniciraju putem mreže, za razliku od klasičnih biblioteka ili modula unutar jednog procesa.
- Servisi su osmišljeni tako da se organiziraju oko **poslovnih funkcionalnosti** ili **domenskih entiteta**. Na primjer, možemo imati zasebne servise za korisnike, proizvode ili narudžbe, pri čemu svaki servis pokriva određeni aspekt poslovanja.
- 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 koda, 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).

> Kada se mikroservisna arhitektura ne koristi na učinkovit način, može doći do nepotrebne složenosti i povećanja kompleksnosti sustava (samim tim i financijskih troškova). Ne tako davni slučaj Elona Muska i Twittera, o kojem se raspravlja u članku na Netokraciji, pruža izvrstan kontekst za razumijevanje kako loša implementacija mikroservisa može rezultirati pretjeranom složenošću, većim troškovima i smanjenom produktivnošću razvojnog tima.

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 u razvoju, održavanju i testiranju. 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. 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 ona to nisu u pravom smislu definicije. 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.

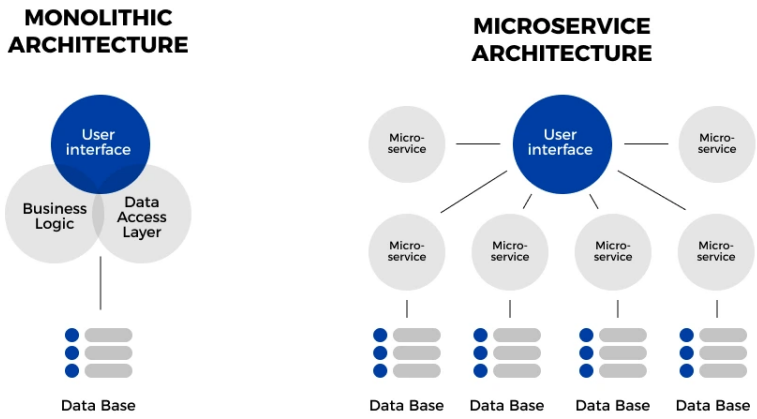
Kako smo na **Programskom inženjerstvu** aplikaciju razvijali u okviru jednog razvojnog okvira (Vue.js), koristeći jedan programski jezik (JavaScript) te koristili Firebase kao servis za autentifikaciju i bazu podataka na način da smo ga integrirali direktno u aplikaciju, možemo argumentirati da smo 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 mikroservisa. Njegova glavna prednost je što eliminira potrebu za brigom o infrastrukturi, upravljanju bazama podataka, skalabilnosti i sličnim tehničkim aspektima, jer sve to rješava sama platforma. Stoga se može reći da ste, na određeni način, svoju aplikaciju razvijali u okviru mikroservisne arhitekture, ali na višem nivou apstrakcije.

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, korisničko sučelje) unutar jednog backend sustava (npr. Express.js).

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

- Što ako broj korisnika aplikacije naglo poraste?
- Kako se učinkovito nositi s velikom količinom podataka u bazi?
- Kako brzo i sigurno isporučiti nove verzije aplikacije korisnicima?
- 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?..

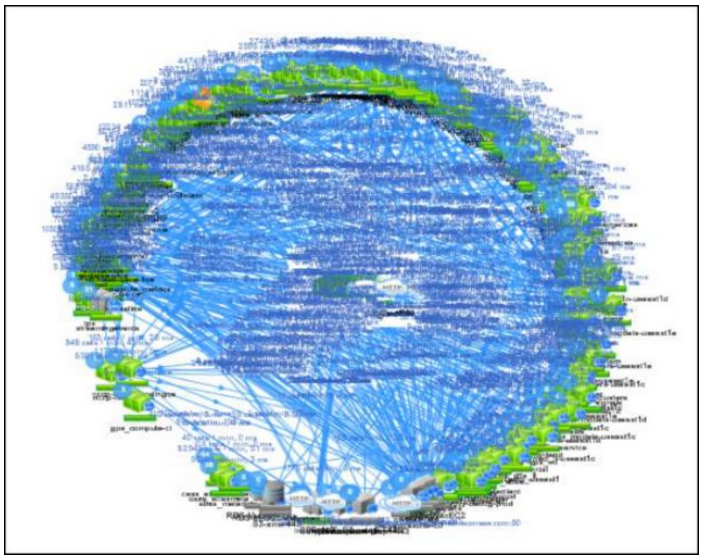
i tako dalje...



> Monolitna vs. mikroservisna arhitektura razvoja aplikacija

Među poznatijim tvrtkama koje uspješno koriste mikroservisnu arhitekturu ističu se **Netflix**, **Amazon**, **Spotify**, **Uber** i **Airbnb**.

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 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.



> Ilustracija mikroservisne arhitekture Netflixa, izvor: zdnet.com

2. Definiranje poslužitelja koristeći `aiohttp`

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 klijenta unutar Pythona koji komunicira s vanjskim servisom**.

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

Prisjetimo se kako definiramo 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()`, kao što smo vidjeli u primjeru slanja konkurentnih zahtjeva na **CatFact API**.

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

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 (host)** i **portu** će poslužitelj slušati.

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

```
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)
```

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.

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

2.3 Definiranje poslužiteljskih ruta

Kako bi poslužitelj bio koristan, odnosno mogao obrađivati nadolazeće zahtjeve, potrebno je definirati **rute** (*eng. route/endpoint*) koje će poslužitelj opsluživati (*eng. serve*). Ruta predstavlja URL putanju koja se koristi za pristup određenom resursu ili funkcionalnosti. Ako još niste, preporuka je da se prisjetite HTTP protokola (skripta `RS4`) kako biste mogli razumjeti gradivo koje slijedi.

2.3.1 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 objekt sadrži sve informacije o zahtjevu, poput: URL putanje, HTTP metode, zaglavlja, tijela zahtjeva i sl.

```
def handler_function(request) # Sinkrona handler funkcija koja prima request objekt
```

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

```
from aiohttp import web

def handler_function(request):
  print(request.method)
  print(request.path)
  print(request.headers)

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)
```

- Ispisuje: GET metodu, URL putanju (`/`), zaglavlja zahtjeva:

```
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 kod, zaglavlja i tijelo odgovora.

Sintaksa `web.Response` 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 kod odgovora (npr. `200`, `404`, `500`)
- `reason` - tekstualni opis statusnog koda (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 naglašavanjem HTTP metode opcijom -X

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

Nakon svake promjene u kodu 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

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 kodu**, 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 i pokreće se iz terminala. Naravno, potrebno je instalirati i Node.js runtime.

```
npm install -g nodemon
```

- ako ne radi, provjerite je li dodan u PATH globalnu varijablu i ponovno pokrenite VS Code/terminal

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 - općeniti 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 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 najčešće korišten za razmjenu podataka između klijenta i poslužitelja.

Rekli smo da format 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:

- **Serijalizacija** - pretvaranje Python objekta u `JSON` format
- **Deserijalizacija** - pretvaranje `JSON` formata u Python objekt

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

Za samo serijalizaciju koristimo metodu `dumps()`:

```
import json

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

json_data = json.dumps(data)

# 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:

```
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**.

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

---

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

2.3.4 POST ruta

Za razliku od GET metode koja se koristi za dohvaćanje podataka, **POST metoda** se 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 najčešće se šalju u `JSON` formatu.

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 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 `json` modul kao kod serijalizacije, već **metodu** `json()` objekta `request`.

```
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)
```

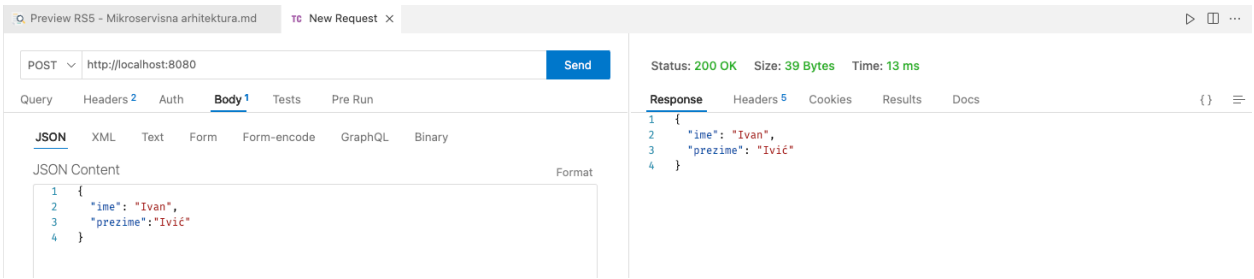
Podatke pošaljite kroz neki od **HTTP klijenata** ili `curl` (`-H` opcija za postavljanje zaglavlja, `-d` opcija za definiranje HTTP tijela):

```
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 poslati kroz HTTP klijent jer ne morate eksplicitno navoditi zaglavlja:



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

---

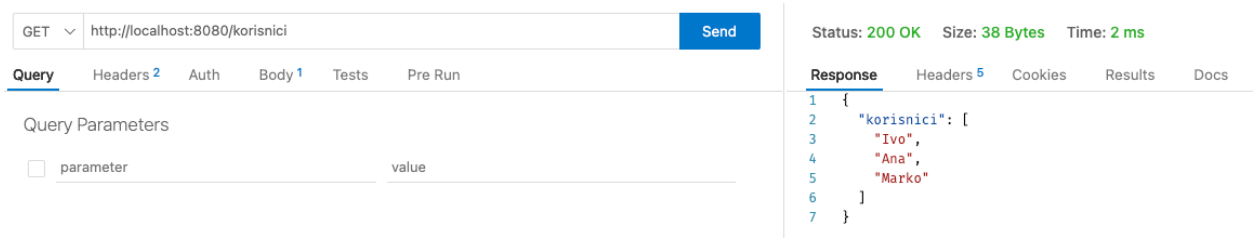
Za dodavanje preostalih HTTP metoda (PUT, DELETE, PATCH) koristimo odgovarajuće ekvivalente na objektu `router`:

- `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**:

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

> Rute možemo definirati na još načina, o tome više u nastavku...

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:8080/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/stariji_korisnici` 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 unutar `aiohttp`

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 koda možemo simulirati komunikaciju između klijenta i poslužitelja** koristeći `aiohttp` klijentsku sesiju i poslužitelj definiran kroz `aiohttp.web` modul.

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

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 kod, 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']})

app = web.Application()

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

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

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

Koji problem uočavate?

▶ Spoiler alert! Odgovor na pitanje

Dakle, problem je što **ako pokrenemo poslužitelj, on će blokirati izvođenje ostatka koda**, uključujući otvaranje klijentske sesije.

Možemo iskoristiti specijalnu Python varijablu `__name__` koja uvijek sadrži naziv trenutačnog modula. Ako pokrenemo skriptu direktno, `__name__` će biti postavljen na `'__main__'`, dok će uvođenjem skripte u drugi modul, `__name__` biti postavljen na naziv modula tog modula.

Preciznije, možemo koristiti `if __name__ == '__main__':` uvjetnu izjavu kako bismo **osigurali da se kod unutar bloka izvršava samo ako je skripta pokrenuta direktno**, a ne uvezena kao modul.

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

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

Primjerice, možemo definirati pokretanje poslužitelja unutar ovog bloka:

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

Ukupan kod:

```
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
```

Kod iznad će svakako prvo otvoriti klijentsku sesiju, obzirom da se `asyncio.run` poziva prije pokretanja poslužitelja. Ako ne bi htjeli 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 onda slati na njega zahtjeve**.

```
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

3.1 `AppRunner` klasa

`AppRunner` klasu koristimo kada nam treba više kontrole nad poslužiteljem, kao što je pokretanje poslužitelja u drugom threadu ili procesu, pokretanje više poslužitelja na različitim adresama i portovima, ili pokretanje poslužitelja na različitim sučeljima.

Glavna prednost `AppRunner` klase je što, za razliku od `web.run_app()` funkcije, **ne blokira izvođenje ostatka koda**, odnosno pruža *non-blocking* način pokretanja poslužitelja, što je ključno kod razvoja raspodijeljenih sustava.

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

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

Kako bismo pokrenuli poslužitelj koristeći `AppRunner` klasu, prvo moramo stvoriti instancu `AppRunner` klase i **registrirati poslužitelj koji želimo pokrenuti**:

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

Postupak je sljedeći:

1. Definiraj `AppRunner` instancu
2. Pokreni `AppRunner` instancu
3. Registriraj poslužitelj
4. Pokreni poslužitelj

Ako je naš poslužitelj definiran lokalno, na portu `8080`, postupak iznad preveden u kod izgleda ovako:

```
from aiohttp.web import AppRunner

runner = AppRunner(app) # 1. Definiraj AppRunner instancu
await runner.setup() # 2. Pokreni AppRunner instancu
site = web.TCPSite(runner, 'localhost', 8080) # 3. Registriraj poslužitelj na adresi
localhost i portu 8080
await site.start() # 4. Pokreni poslužitelj
```

Sintaksa:

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

> Ova 4 koraka gotovo uvijek će se ponavljati pa ih je praktično spakirati u zasebnu korutinu `start_server` ili `run_server`

```
async def start_server():
  runner = AppRunner(app)
  await runner.setup()
  site = web.TCPSite(runner, "localhost", 8080)
  await site.start()

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

Sada imamo dvije korutine, `main` i `start_server`, koje želimo pokrenuti. Međutim, rekli smo da s `asyncio.run` možemo pokrenuti samo jednu korutinu.

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

```
async def main():
  await start_server()
  ...

asyncio.run(main())
```

Ukupan kod:

```
from aiohttp import web
from aiohttp.web import AppRunner
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 = web.TCPSite(runner, 'localhost', 8080)
  await site.start()
  print("Poslužitelj sluša na http://localhost:8080")

async def main():
  await start_server() # Prvo pokreni poslužitelj
  async with aiohttp.ClientSession() as session: # Zatim otvori klijentsku sesiju
    print("Klijentska sesija otvorena")
    pass

asyncio.run(main()) # Pokreni main korutinu
```

Ispisuje:

```
Poslužitelj sluša na http://localhost:8080
Klijentska sesija otvorena
```

Kako ćemo sada napokon poslati zahtjev na ovaj poslužitelj koristeći klijentsku sesiju?

▶ Spoiler alert! Odgovor na pitanje

```
async def main():
  await start_server() # Prvo pokreni poslužitelj
  async with aiohttp.ClientSession() as session: # Zatim otvori klijentsku sesiju
    rezultat = await session.get('http://localhost:8080/korisnici') # Pošalji GET zahtjev
na lokalni poslužitelj
    print(await rezultat.text()) # Ispis odgovora
```

Kad pokrenemo kod, 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:

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

Važno! Ako pokušate ponovno poslati zahtjev direktno iz terminala ili kroz HTTP klijent, dobit ćete grešku zato što poslužitelj više ne radi (jednom kad se završi main korutina, poslužitelj se gasi). Moguće je stvari riješiti beskonačnim petljama ako bi to baš htjeli, ali to **nije preporučeno**.

> Puno bolji pristup je, odvojiti poslužitelja i klijentsku sesiju u zasebne skripte, no o tome više u nastavku...

3.2 GET ruta s URL parametrima

Kroz nekoliko primjera ćemo pokazati sve što smo do sad naučili, preciznije, vidjet ćemo kako konkurentno slati HTTP zahtjeve definiranjem klijentskih sesija na interne poslužitelje.

Do sad smo definirali jedan poslužitelj, međutim moguće ih je unutar jedne skripte definirati i više.

Uobičajeno je kada šaljemo HTTP odgovor unutar *handler funkcije*, koristiti `web.json_response()` funkciju te definirati statusni kod 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, route parametar 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 kod 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`:

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

Primjer slanja 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'}
```

3.3 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 kodom.

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.

---

to be continued...