

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(4) Asinkroni Python: Konkurentna obrada mrežnih operacija

Na prethodnim vježbama upoznali smo temeljne koncepte asinkronog programiranja u Pythonu koristeći biblioteku `asyncio`. Naučili smo kako definirati i pokretati korutine, konkurentno izvršavati kod pomoću `asyncio.gather` i `asyncio.create_task`, kao i ulogu *event loop* mehanizma koji omogućava učinkovito izvršavanje korutina unutar jedne Python dretve - samu konkurentnost.

Istaknuli smo kako I/O operacije često predstavljaju usko grlo u programima jer zahtijevaju čekanje na vanjske resurse, primjerice mrežne odgovore, pristup datotekama ili komunikaciju s bazama podataka. Učinkovito upravljanje takvim operacijama kroz konkurentno izvođenje ključno je za razvoj raspodijeljenih sustava, u kojima se funkcionalnosti nalaze na više čvorova međusobno povezanih mrežom.

Do sada smo sve primjere temeljili na simuliranim I/O operacijama koristeći `asyncio.sleep`, što nam je omogućilo razumijevanje osnovnih principa asinkronog programiranja. Na ovim vježbama ćemo se fokusirati na praktičnu primjenu asinkronog programiranja i konkurentne obrade u kontekstu stvarnih I/O operacija - **obrada mrežnih zahtjeva kroz HTTP protokol**.

Posljednje ažurirano: 20.11.2025.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(4\) Asinkroni Python: Konkurentna obrada mrežnih operacija](#)
 - [Sadržaj](#)
 - [1. HTTP \(HyperText Transfer Protocol\) protokol](#)
 - [1.1 Osnove HTTP protokola](#)
 - [1.1.1 Struktura HTTP zahtjeva](#)
 - [1.1.2 Struktura HTTP odgovora](#)
 - [2. Konkurentna obrada mrežnih operacija pomoću aiohttp biblioteke](#)

- [2.1 Kako šaljemo HTTP zahtjeve sinkrono \(`requests`\)?](#)
 - [2.2 Kako šaljemo HTTP zahtjeve konkurentno \(`aiohttp`\)?](#)
 - [2.2.1 Context Manager `with`](#)
 - [2.2.2 `clientSession` klasa](#)
 - [2.2.3 Konkurentna obrada HTTP zahtjeva \(`asyncio.gather`\)](#)
 - [2.2.4 Konkurentna obrada HTTP zahtjeva \(`asyncio.Task`\)](#)
 - [2.3 Timeout mrežnih operacija i obrada iznimki](#)
- [3. Zadaci za vježbu - Konkurentna obrada mrežnih operacija i simulacije grešaka](#)
 - [Zadatak 1: `fetch_users` i izdvajanje podataka](#)
 - [Zadatak 2: `filter_cat_facts`](#)
 - [Zadatak 3: `mix_dog_cat_facts`](#)
 - [Zadatak 4: simulacija autentifikacije korisnika](#)
 - [Zadatak 5: Pretvorba sinkronog koda u asinkroni](#)
 - [Zadatak 6: Simulacija raspodijeljenog sustava za dohvaćanje i obradu vremenskih podataka](#)

1. HTTP (HyperText Transfer Protocol) protokol

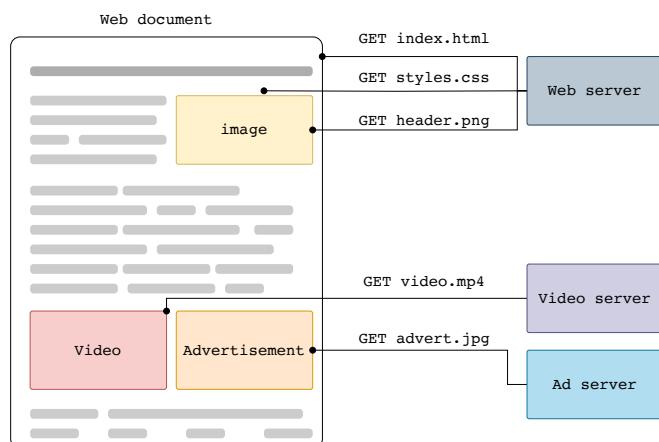
HTTP protokol je dominantan mrežni protokol na aplikacijskoj razini (*application-level protocol*) koji se koristi u modernim web aplikacijama i *web-facing* raspodijeljenim sustavima. Predstavlja jednostavan, ali snažan *real-world* mehanizam za razmjenu podataka između nezavisnih čvorova u sustavu.

HTTP je **request-response** protokol osmišljen prema modelu *klijent-poslužitelj* ([client-server model](#)).

- **Klijent** (eng. *client*) inicira komunikaciju slanjem HTTP zahtjeva poslužitelju radi dohvatanja, slanja ili izmjene resursa.
- **Poslužitelj** (eng. *server*) prima HTTP zahtjev, obrađuje ga i vraća HTTP odgovor klijentu s traženim podacima ili statusom operacije.

Podsjetnik: Na kolegiju web aplikacije, klijent nam je bila Vue.js web aplikacija koja se izvodi u web pregledniku, dok je poslužitelj bio Node.js Express server koji je obrađivao HTTP zahtjeve i komunicirao s bazom podataka MongoDB. U kontekstu raspodijeljenih sustava, **klijent i poslužitelj mogu biti bilo koji čvorovi u mreži koji koriste HTTP za međusobnu komunikaciju** - nismo više ograničeni na klasični klijent-poslužitelj model.

HTTP je u [svojim počecima](#) (90-ih) bio namijenjen isključivo za prijenos hipertekstualnih dokumenata (HTML stranica) između web preglednika i web poslužitelja. Međutim, s vremenom je evoluirao u univerzalni protokol za komunikaciju između različitih vrsta *service-oriented* arhitektura, evoluirajući u industrijski standard *general-purpose application-layer* mrežnog protokola za **prijenos gotovo svih vrsta podataka**, uključujući JSON, XML, slike, audio, video i druge binarne formate.



HTTP zahtjevima mogu se "dohvatiti" preko mreže razni resursi s različitih poslužitelja

Zašto učimo HTTP u kontekstu raspodijeljenih sustava?

Većina suvremenih raspodijeljenih sustava, uključujući mikroservise (AWS, Azure Functions, Google Cloud Functions), REST API sučelja, *serverless* arhitekture i web servise, koristi HTTP kao primarni komunikacijski protokol. HTTP je **široko prihvaćen, implementiran** i detaljno **dokumentiran** u gotovo svim modernim programskim jezicima, operacijskim sustavima, platformama i cloud okruženjima.

Primjeri korištenja HTTP protokola u raspodijeljenim sustavima:

- mikroservisi često koriste REST ili gRPC sučelja preko HTTP/2 protokola
- sve moderne cloud platforme (AWS, Azure, Google Cloud) nude HTTP/HTTPS *endpointove* za svoje funkcije i servise
- *serverless* "okidači" često se temelje na HTTP zahtjevima
- IoT platforme redovito koriste HTTPS za sigurno upravljanje uređajima
- web aplikacije i moderne SPA (*Single Page Applications*) koriste HTTP za komunikaciju s *backend* servisima

Naravno, **HTTP nije jedini protokol koji se koristi u raspodijeljenim sustavima.** *Domain-specific* protokoli često su bolje prilagođeni određenim scenarijima, ali su često i složeniji za kvalitetnu implementaciju i održavanje. Primjeri drugih protokola uključuju:

- *message Queue/Event Streams* protokoli poput AMQP (RabbitMQ), MQTT (dominantan u IoT sustavima) i Kafka protokola (Apache Kafka)
- baze podataka gotovo uvijek koriste vlastite protokole za komunikaciju klijenta i poslužitelja (npr. PostgreSQL, MySQL, MongoDB)
- RPC ([remote procedure call](#)) protokoli poput gRPC (koji koristi HTTP/2 kao transport), ili Thrift (koji podržava različite transportne mehanizme)
- protokoli niske latencije, poput WebSocket-a za dvosmjernu komunikaciju u stvarnom vremenu, te razni *custom UDP-based* protokoli u *real-time* aplikacijama, *gaming* poslužiteljima, streaming platformama i sl.
- distribuirani klasteri i sustavi za pohranu često koriste vlastite protokole za internu komunikaciju (npr. Cassandra, Hadoop HDFS)

HTTP je *stateless* protokol

HTTP je [*stateless protokol*](#) ("ne pamti stanje prethodnih zahtjeva"), što je iznimno pogodno za učenje raspodijeljenih sustava u kojima su neovisnost čvorova i idempotentnost operacija ključni koncepti. [Idempotentnost](#) znači da se **višestrukim izvršavanjem iste operacije postiže isti rezultat kao i jednim pozivom**. U HTTP kontekstu to podrazumijeva da ponavljanje istog zahtjeva neće promijeniti stanje resursa nakon prvog izvršenja — iako to **ovisi o korištenoj HTTP metodi** (npr. `GET` i `DELETE` su idempotentne, dok `POST` nije).

Otpornost na greške ([eng. Fault tolerance](#)) jednako je važna u raspodijeljenim sustavima. HTTP, kao i mnogi drugi protokoli, nudi mehanizme za signaliziranje grešaka putem statusnih kodova, čime omogućuje klijentima pravovremenu reakciju i oporavak od privremenih problema u mreži ili na poslužitelju.

Analiza HTTP zahtjeva pokazuje gdje se u stvarnim I/O operacijama javljaju uska grla

U radu s mrežnim operacijama često se susrećemo s uskim grlima koja mogu znatno utjecati na performanse. Mrežne operacije su sporije od operacija u memoriji ili na disku zbog latencije, ograničene propusnosti, zagušenja mreže i drugih faktora (više o tome na predavanjima iz kolegija).

Ključno je razumjeti da se **najveća uska grla** (eng. *bottlenecks*) u raspodijeljenim sustavima **gotovo uvijek javljaju u mrežnoj komunikaciji**, a ne u računalnoj snazi (*computing resources* - CPU, RAM, disk).

HTTP zahtjevi nam omogućuju simulaciju sljedećeg:

- **RPC-like poziva:** izvršavanje funkcija na udaljenim poslužiteljima (temelj mnogih mikroservisnih arhitektura)
- **sporih i nepouzdanih mrežnih uvjeta:** latencije, gubitka paketa, ograničene propusnosti i drugih realnih mrežnih karakteristika
- **komunikacije i međuvisnosti između servisa:** testiranje ponašanja složenih sustava sastavljenih od više međusobno povezanih servisa

Ovo čini HTTP idealnim za učenje i demonstraciju koncepata poput:

- konkurentnosti
- asinkronog programiranja
- *timeout* i *retry* mehanizama
- *fault tolerance*
- propagacije kvarova (*failure propagation*)

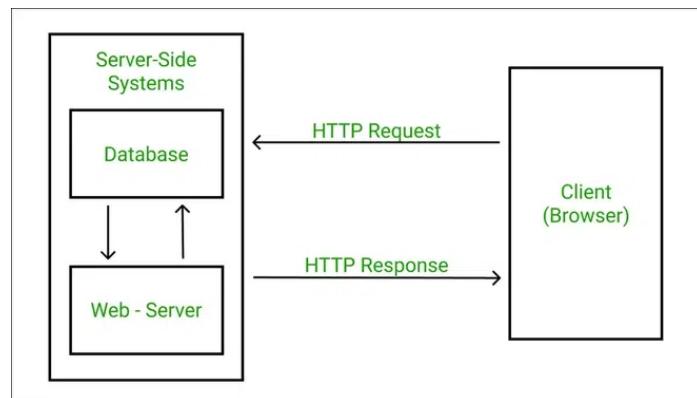
Zaključno: HTTP predstavlja jednostavan i nemetljiv protokol, lako razumljiv te pogodan za implementaciju u raznim programskim jezicima, a pritom osigurava jasan i predvidljiv komunikacijski model između klijenta i poslužitelja. U kontekstu raspodijeljenih sustava, preporučljivo je započeti upravo s HTTP-om prije razmatranja složenijih rješenja poput gRPC-a, WebSocketa ili Raft RPC-a.

1.1 Osnove HTTP protokola

Tipičan HTTP komunikacijski model (**klijent ↔ poslužitelj**) temelji se na razmjeni HTTP zahtjeva i HTTP odgovora.

HTTP zahtjev (eng. *HTTP request*) predstavlja poruku koju klijent upućuje poslužitelju radi dohvatanja, slanja ili izmjene resursa. Primjerice, web preglednik šalje HTTP zahtjev za određeni resurs udaljenom poslužitelju, npr. HTTP zahtjev za izvršavanje bankovne transakcije prema API-ju *transaction-service* mikroservisa.

HTTP odgovor (eng. *HTTP response*) predstavlja poruku koju poslužitelj vraća klijentu kao rezultat obrade zahtjeva. Primjerice, poslužitelj može vratiti HTTP odgovor s JSON podacima u tijelu odgovora, npr. JSON HTTP odgovor s podacima o autentificiranom korisniku.



HTTP komunikacijski model: klijent šalje HTTP zahtjev poslužitelju, koji obrađuje zahtjev i vraća HTTP odgovor klijentu.

1.1.1 Struktura HTTP zahtjeva

- **Metoda** (eng. *method*): označava tip operacije koju klijent želi izvršiti nad resursom (npr. `GET`, `POST`,

`PUT`, `PATCH`, `DELETE`).

- **URL** (*eng. Uniform Resource Locator*): specificira lokaciju resursa na poslužitelju (npr.

`https://api.github.com/users/nek_i_korisnik`). Sastoji se od sljedećih komponenti:

- **Shema** (*eng. scheme*): definira protokol komunikacije (npr. `https`).
 - **Domena** (*eng. domain*): predstavlja naziv poslužitelja (npr. `api.github.com`).
 - **Route parametar** (*eng. route parameter*): opisuje dinamički dio putanje koji često identificira pojedini resurs (npr. `/users/:id`).
 - **Query parametar** (*eng. query parameter*): dodatan parametar upita koji se koristi za filtriranje, paginaciju, sortiranje i sl. (npr. `?page=1&limit=10`).
 - **Fragment** (*eng. fragment*): označava dio resursa, najčešće za navigaciju unutar dokumenta (npr. `#section1`).
- **Zaglavljia** (*eng. headers*): pružaju meta-informacije o zahtjevu, poput tipa sadržaja, autentikacije ili preferiranog formata odgovora (npr. `Content-Type: application/json`).
 - **Tijelo** (*eng. body*): sadrži podatke koje klijent šalje poslužitelju, najčešće u formatima poput JSON-a, XML-a ili form-data; koristi se primarno u metodama `POST`, `PUT` i `PATCH`.
 - **Verzija protokola** (*eng. protocol version*): označava verziju HTTP protokola korištenu u zahtjevu (npr. `HTTP/1.1`, `HTTP/2`, `HTTP/3`).



Struktura HTTP zahtjeva: metoda, URL, zaglavljia i tijelo zahtjeva.

Najčešće korištene HTTP metode imaju jasno definirane semantike i koriste se u skladu s **namjerom operacije nad resursom**:

- **GET**: dohvata jedan ili više resursa s poslužitelja (npr. podatke o korisniku).
- **POST**: šalje podatke na poslužitelj radi stvaranja novog resursa ili obrade poslanih podataka (npr. slanje podataka iz forme).
- **PUT**: u potpunosti zamjenjuje postojeći resurs novim podatcima (npr. ažuriranje kompletног profila korisnika).
- **PATCH**: djelomično ažurira postojeći resurs (npr. promjena lozinke ili ažuriranje samo jednog polja).
- **DELETE**: trajno uklanja resurs s poslužitelja (npr. brisanje korisnika).

Dodatna pojašnjenja i primjeri HTTP zahtjeva

GET metoda

Koristi se isključivo za dohvaćanje podataka. Često se kombinira s **query parametrima** (filtriranje, sortiranje, paginacija) ili s **route parametrima** kada je potrebno dohvatiti konkretni resurs (npr. korisnika prema ID-u). Važno je naglasiti da slanje **tijela zahtjeva** unutar GET metode nije u skladu sa standardom i općenito se ne prakticira.

POST metoda

Primjenjuje se kada je potrebno poslati podatke na poslužitelj, najčešće radi stvaranja novog resursa ili obrade ulaznih podataka. Podaci se prosljeđuju u **tijelu zahtjeva**, obično u JSON formatu, što omogućuje slanje kompleksnih struktura koje ne bi bilo prikladno slati putem query parametara. Prednost je i to što se podaci ne prikazuju u URL-u.

PUT i PATCH metode

Obje se koriste za ažuriranje postojećeg resursa, no semantika se razlikuje:

- **PUT** zahtjeva slanje **cjelovitog** skupa podataka resursa te u pravilu zamjenjuje postojeći resurs novim.
- **PATCH** omogućuje **djelomičnu izmjenu**, pa se šalju samo ona polja koja je potrebno ažurirati.

U oba slučaja, podaci za ažuriranje šalju se u **tijelu zahtjeva**, najčešće u JSON formatu.

Pokazat ćemo nekoliko primjera slanja HTTP zahtjeva koristeći CLI alat `curl`.

Sintaksa:

```
→ curl -X <METODA> "<URL>" [DODATNE_OPCIJE]
→ curl -X <METODA> "<URL>" -H "Content-Type: application/json" -d '<PODACI>'
```

Primjer: HTTP zahtjev koji dohvaća podatke o korisniku `pero_peric`:

```
→ curl -X GET "https://api.github.com/users/pero_peric"
```

Pokušajte poslati zahtjev u terminalu s vašim GitHub korisničkim imenom kako biste dobili HTTP odgovor javno dostupnih podataka o vašem profilu.

Primjer: HTTP zahtjev koji šalje podatke o novom korisniku na poslužitelj:

- Opcija `-d` označava podatke koji se šalju u tijelu zahtjeva
- Opcija `-H` označava zaglavljve zahtjeva
- Zaglavljve `Content-Type: application/json` specificira da su podaci u JSON formatu
- Oznakom `\` možemo razbiti naredbu na više redaka radi bolje čitljivosti

```
→ curl -X POST "https://api.github.com/users" \
-H "Content-Type: application/json" \
-d '{
    "username": "pero_peric",
    "email": "pperic@gmail.com",
    "password": "pero123"
}'
```

Napomena: Primjeri su ilustrativni; GitHub API ne podržava stvaranje korisnika putem javnog API-ja, niti brisanje/ažuriranje postojećih korisnika.

Primjer: HTTP zahtjev koji ažurira username korisnika `pero_peric`:

```
→ curl -X PATCH "https://api.github.com/users/pero_peric" \
-H "Content-Type: application/json" \
-d '{
    "username": "pero_peric_2"
}'
```

Primjer: HTTP zahtjev koji briše korisnika `pero_peric`:

```
→ curl -X DELETE "https://api.github.com/users/pero_peric"
```

Primjer: HTTP zahtjev koji dohvaća samo korisnike s imenom `pero`:

```
→ curl -X GET "https://api.github.com/users?name=pero"
```

Primjer: HTTP zahtjev koji zamjenjuje sve podatke o korisniku `pero_peric`:

```
→ curl -X PUT "https://api.github.com/users/pero_peric" \
-H "Content-Type: application/json" \
-d '{
    "username": "pero_peric_2",
    "email": "pperic2@gmail.com",
    "password": "pppper01234"
}'
```

Iz navedenih primjera uočite dijelove HTTP zahtjeva: metodu, URL, zaglavlja i tijelo zahtjeva.

1.1.2 Struktura HTTP odgovora

- **Statusna linija** (*eng. status line*): uključuje **statusni kod** i pripadajuću **tekstualnu poruku** (npr. `200 OK`), pri čemu brojčana vrijednost (`200`) označava rezultat obrade zahtjeva, a tekstualni opis (`OK`) pruža kratko objašnjenje stanja.
- **Zaglavlja** (*eng. headers*): pružaju **dodatne informacije o odgovoru**, poput formata podataka, duljine sadržaja ili informacija o keširanju (npr. `Content-Type: application/json`).
- **Tijelo** (*eng. body*): sadrži **podatke koje poslužitelj vraća klijentu**, najčešće u formatima kao što su

JSON, HTML ili XML.

- **Verzija protokola** (*eng. protocol version*): specificira **verziju HTTP protokola** korištenu u odgovoru (npr. `HTTP/1.1`).

RESPONSE



Struktura HTTP odgovora: statusna linija, zaglavla i tijelo odgovora.

Statusni kodovi

Statusni kodovi (*eng. HTTP status codes*) koriste se u HTTP odgovorima kako bi klijent dobio jasnu informaciju o rezultatu obrađenog zahtjeva. Ako je zahtjev neispravan ili dođe do pogreške, poslužitelj vraća odgovarajući statusni kod koji opisuje vrstu problema.

Statusne kodove možemo grupirati u sljedeće kategorije:

- **1xx (100-199) – Informacijski odgovori** (*eng. Informational responses*)
Poslužitelj je primio zahtjev i nastavlja s njegovom obradom.
- **2xx (200-299) – Uspješni odgovori** (*eng. Successful responses*)
Zahtjev je uspješno primljen, razumljen i obrađen.
- **3xx (300-399) – Preusmjeravanja** (*eng. Redirection messages*)
Klijent mora poduzeti dodatne korake kako bi dovršio zahtjev, najčešće slijediti novu lokaciju resursa.
- **4xx (400-499) – Klijentske pogreške** (*eng. Client error responses*)
Zahtjev sadrži pogreške na strani klijenta (npr. neispravni podaci, nedostatna autorizacija).
- **5xx (500-599) – Poslužiteljske pogreške** (*eng. Server error responses*)
Poslužitelj je naišao na pogrešku prilikom obrade valjanog zahtjeva.

Statusni kodovi standardiziraju komunikaciju između klijenta i poslužitelja, omogućujući klijentu da interpretira odgovor i poduzme odgovarajuće akcije ovisno o rezultatu.

Popis svih statusnih kodova i njihova objašnjenja dostupan je na službenoj [Mozilla dokumentaciji](#).

Najčešće korišteni statusni kodovi

- **200 OK** – Zahtjev je uspješno obrađen (npr. dohvata resursa putem GET zahtjeva)
- **201 Created** – Novi resurs je uspješno stvoren (npr. nakon POST zahtjeva)
- **400 Bad Request** – Zahtjev nije moguće obraditi zbog neispravnih ili nedostajućih podataka (npr.

neispravan JSON format)

- **404 Not Found** – Traženi resurs ne postoji na poslužitelju (primjerice, pogrešan URL s dinamičkim parametrom).
- **500 Internal Server Error** – Opća pogreška na poslužitelju, tipično uzrokovana greškom u kodu (iznimka na poslužitelju koja nije "uhvaćena")
- **401 Unauthorized** – Klijent nije autentificiran, te nema pristup resursu.
- **204 No Content** – Zahtjev je uspješno obrađen, ali odgovor nema tijelo (npr. nakon uspješnog brisanja).
- **403 Forbidden** – Klijent je autentificiran, ali nema odgovarajuća prava pristupa.
- **301 Moved Permanently** – Resurs je trajno premješten na novu adresu.
- **503 Service Unavailable** – Poslužitelj trenutno nije dostupan (npr. zbog preopterećenja ili održavanja).
- **409 Conflict** – Zahtjev nije moguće obraditi zbog konflikta u stanju resursa (npr. prilikom ažuriranja zastarjelih podataka).

Napomena: Studentima kojima sažetak iz ovog poglavlja nije dovoljan za jasno razumijevanje HTTP protokola preporučuje se da dodatno prouče skriptu WA1 iz kolegija Web aplikacije, dostupnu na [GitHubu kolegija](#).

2. Konkurentna obrada mrežnih operacija pomoću aiohttp biblioteke

aiohttp (Asynchronous HTTP Client/Server for `asyncio` and Python) je popularna Python biblioteka koja omogućuje **asinkrono programiranje HTTP klijenata i poslužitelja u Pythonu**. Ova datoteka razvijena je na temelju `asyncio` biblioteke s kojom smo se već upoznali u skripti RS3.

aiohttp biblioteka omogućuje nam da jednostavno implementiramo asinkrone HTTP **klijente i poslužitelje** u Pythonu, što je korisno u kontekstu razvoja i testiranja malih web servisa koji zahtijevaju visoku propusnost. Dodatno, datoteka pruža podršku za [WebSocket protokol](#).

Biblioteka `aiohttp` koristi `asyncio` event loop za upravljanje asinkronim operacijama, što omogućuje učinkovito rukovanje velikim brojem istovremenih HTTP zahtjeva bez potrebe za višedretvenim pristupom. U usporedbi s tradicionalnim sinkronim klijentima, poput `requests`, `aiohttp` omogućuje konkurentno slanje HTTP zahtjeva i obradu odgovora, čime se značajno poboljšavaju performanse u scenarijima s velikim brojem mrežnih operacija.

Dakle, `aiohttp` je *non-blocking asinkrona I/O biblioteka*, baš kao i `asyncio`.

Instalacija aiohttp biblioteke

Za razliku od `asyncio` biblioteke koja je ugrađena od Python 3.7+, `aiohttp` biblioteku potrebno je instalirati ručno:

```
→ pip install aiohttp
```

Napomena: ako koristite `macos` ili `Linux`, vjerojatno ćete trebati koristiti `pip3` umjesto `pip`, ako niste podesili alijase ili druge konfiguracijske postavke iz RS1.

Kod instalacije vanjskih paketa **preporučuje se korištenje virtualnog okruženja** kako bi se izbjegli konflikti između paketa i "krcanje" globalne Python distribucije.

Ako ste se odlučili koristiti `conda` alat za upravljanje virtualnim okruženjima, stvorite novo virtualno okruženje naziva `rs4` prije nego instalirate `aiohttp` biblioteku:

```
→ conda create --name rs4 python=3.13
```

Aktivirajte okruženje:

```
→ conda activate rs4
```

Unutar VS Codea promijenite *interpreter* na novo kreirano okruženje `rs4` kako biste izbjegli [linting greške](#).

```
→ CTRL/CMD + SHIFT + P → Python: Select Interpreter → rs4
```

Sada možete instalirati biblioteke 

2.1 Kako šaljemo HTTP zahtjeve sinkrono (`requests`)?

Međutim, prije nego se upoznamo s asinkronim načinom definiranja HTTP klijenata, vrijedno je prisjetiti se kako to radimo sinkrono, koristeći biblioteku `requests`.

`requests` je popularna biblioteka za rad s HTTP zahtjevima u Pythonu koja omogućuje jednostavno slanje zahtjeva na poslužitelj i primanje odgovora. Međutim, `requests` je **sinkrona biblioteka**, što znači da će svaki zahtjev blokirati izvođenje programa dok se ne primi odgovor.

Kako bismo poslali HTTP zahtjev koristeći `requests` biblioteku, prvo je potrebno instalirati biblioteku:

```
pip install requests
```

Uključimo `requests` biblioteku:

```
import requests
```

Jednostavni primjer slanja GET zahtjeva na poslužitelj. Zahtjev ćemo poslati na [Cat Facts API](#) servis koji vraća nasumične činjenice o mačkama:

Sintaksa:

```
requests.HTTP_METHOD("<URL>")
```

- `HTTP_METHOD` predstavlja HTTP metodu koju želimo koristiti (npr. `GET`, `POST`, `PUT`, `DELETE`).
- `<URL>` je URL adresa na koju šaljemo zahtjev.
- *Primjer HTTP GET:* `requests.get("https://catfact.ninja/fact")` šalje GET zahtjev na navedeni URL.
- *Primjer HTTP POST:* `requests.post("https://example.com/api", json=python_dict)` šalje POST zahtjev s podacima u tijelu zahtjeva.

Primjer slanja GET zahtjeva na endpoint `/fact`:

URL servisa: <https://catfact.ninja>

```
import requests

response = requests.get("https://catfact.ninja/fact") # HTTP GET zahtjev
print(response.text) # ispis tijela HTTP odgovora
```

Ako pokrenemo ovaj kod, dobit ćemo nasumični odgovor u obliku rječnika s ključevima `fact` i `length`:

```
{
    "fact": "The life expectancy of cats has nearly doubled over the last fifty years.",
    "length": 73
}
```

Možemo provjeriti **statusni kod** odgovora:

```
print(response.status_code) # 200
```

Možemo i deserijalizirati JSON odgovor koristeći metodu `json()`:

Zapamti: [Serijalizacija](#) podataka predstavlja proces pretvaranja podataka u format pogodan za prijenos ili pohranu (npr. JSON string). Deserijalizacija (*unserialization/unmarshalling*) je obrnuti proces, gdje se serijalizirani podaci pretvaraju natrag u format pogodan za računalnu obradu (npr. Python rječnik).

```
data = response.json() # deserijalizacija JSON odgovora u Python rječnik
print(data["fact"]) # ispis samo činjenice o mačkama
```

Testirajte slanje nekoliko zahtjeva na isti API kako biste vidjeli različite činjenice o mačkama.

Pitanje: Kako ćete poslati zahtjev koristeći alat `curl` iz terminala?

► Spoiler alert! Odgovor na pitanje

Rekli smo da u sinkronim programima, svaka funkcija koju pozovemo, ako traje dugo, blokira izvođenje programa dok se ne završi. To vrijedi i za HTTP zahtjeve.

- u sinkronoj obradi, svaki zahtjev koji pošaljemo čeka na odgovor prethodnog prije nego pošaljemo novi. Ako neki zahtjevi traju dugo, to može značajno usporiti izvođenje programa.

Primjer: Poslat ćemo pet zahtjeva na Cat Facts API. Kod za slanje možemo staviti u jednostavnu funkciju koja šalje GET zahtjev i ispisuje rezultat. Sam podatak možemo dohvatiti preko ključa `fact`, ali prije toga moramo **deserializirati JSON odgovor** pomoću metode `json()`.

```
import requests

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"]
    print(fact)

print("Šaljemo 1. zahtjev...")
send_request()

print("Šaljemo 2. zahtjev...")
send_request()

print("Šaljemo 3. zahtjev...")
send_request()

print("Šaljemo 4. zahtjev...")
send_request()

print("Šaljemo 5. zahtjev...")
send_request()
```

Vidimo da je za izvršavanje svakog zahtjeva potrebno pričekati odgovor prethodnog; na taj način smo napisali kod i to je OK.

Ukupno vrijeme trajanja ovog programa je prosječno 1-2 sekunde, ovisno prvenstveno o brzini interneta i opterećenju servisa Cat Facts API.

Možemo koristiti biblioteku `time` kako bismo preciznije izmjerili vrijeme izvršavanja programa:

```
import requests
import time

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"] # ovo ne možemo raditi s aiohttp varijantom response
    # objekta budući da ".json()" postaje asinkrona metoda
    print(fact)

start = time.perf_counter()
```

```

print("Šaljemo 1. zahtjev...")
send_request()

print("Šaljemo 2. zahtjev...")
send_request()

print("Šaljemo 3. zahtjev...")
send_request()

print("Šaljemo 4. zahtjev...")
send_request()

print("Šaljemo 5. zahtjev...")
send_request()

end = time.perf_counter()
print(f"Izvršavanje programa traje {end - start:.2f} sekundi.")

```

Poslat ćemo 15 zahtjeva, a kod ćemo strukturirati u `for` petlju:

```

import requests
import time

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"] # ovo ne možemo raditi s aiohttp varijantom response
    # objekta budući da ".json()" postaje asinkrona metoda
    print(fact)

start = time.perf_counter()

for i in range(15):
    print(f"Šaljemo {i + 1}. zahtjev...")
    send_request()

end = time.perf_counter()
print(f"Izvršavanje programa traje {end - start:.2f} sekundi.")

```

Prosječno vrijeme trajanja programa iznad je 3-4 sekunde. Ako povećamo broj zahtjeva, **vrijeme izvršavanja će se povećati proporcionalno broju zahtjeva.**

Obzirom da vrijeme izvođenja programa direktno ovisi o broju iteracija `i`, možemo reći da je vremenska složenost $O(n)$. Ipak, HTTP zahtjevi su **I/O operacije** i u praksi je teško predvidjeti točno vrijeme trajanja programa zbog varijabilnih mrežnih uvjeta, ali kod analize kompleksnosti algoritma, prepostavljamo idealne uvjete, odnosno tretiramo mrežne operacije kao *constant-time* operacije.

Zahtjeve smo do sada slali **sinkrono**, što je jasno vidljivo u ovom primjeru. Zamislimo situaciju u kojoj šaljemo 1 000 ili čak 10 000 zahtjeva – program bi tada radio izrazito dugo, a naš servis bi morao čekati da se svaki zahtjev pojedinačno obradi.

U nastavku ćemo vidjeti kako ćemo ovo riješiti asinkronim programiranjem odnosno **konkurentnim slanjem i obradom HTTP zahtjeva** na poslužitelj pomoću `aiohttp` biblioteke.

Napomena: ako pošaljete previše zahtjeva, moguće da će CatFact API blokirati zahtjeve s vaše IP adresu, te da će vam se vratiti statusni kod 429 (Too Many Requests). Možete pokušati ponovo kasnije ili koristiti alternativni API za testiranje

[Lista dostupnih besplatnih API-ja za testiranje](#) - odaberite API bez CORS ili API-key ograničenja.

2.2 Kako šaljemo HTTP zahtjeve konkurentno (aiohttp)?

Cilj nam je poslati više zahtjeva na *Cat Facts API* i postići brže vrijeme izvršavanja programa (ne želimo da slanje i ispis rezultata 15 zahtjeva traje gotovo 4 sekunde).

Kada razmišljamo o konkurentnom slanju HTTP zahtjeva, najidealnije je razmišljati o **korutinama**. Na zadnjim vježbama smo definirali korutine s fiksnim čekanjima (*non-blocking* `asyncio.sleep`), a sada ćemo definirati korutine koje će slati HTTP zahtjeve na neki servis.

Kako zahtjeve šaljemo konkurentno, najpraktičnije je kod spakirati u korutine.

```
import aiohttp # slanje HTTP zahtjeva
import asyncio # temelj asinkronog programiranja - rad s event loopom
import time # mjerjenje vremena izvršavanja
```

Nećemo više koristiti *blocking* `requests` biblioteku, zamijenili smo je s *non-blocking* `aiohttp`.

Biblioteka `requests` u pozadini uspostavlja korisničku sesiju (*client session*) koja omogućuje **ponovnu upotrebu veze s poslužiteljem** te pohranu HTTP zaglavlja, podataka o autentifikaciji, kolačića i drugih elemenata koji se ponavljaju pri svakom HTTP zahtjevu. Zahvaljujući tome, umjesto stvaranja nove sesije za svaki pojedini zahtjev, **moguće je više puta koristiti već uspostavljenu sesiju**.

U `aiohttp` biblioteci, potrebno je naglasiti **definiranje asinkrone sesije** - ona nam omogućuje iste funkcionalnosti koje su prethodno navedene.

2.2.1 Context Manager with

Koncept **kontekstnog menadžera** (eng. *Context Manager*) u Pythonu omogućavaju nam alokaciju i dealokaciju resursa, odnosno upravljanje resursima koji se koriste u bloku koda.

Najčešće korišteni primjer *context managera* u Pythonu je naredba `with` koju koristimo kako bismo definirali **blok koda za rad s resursima** koje treba eksplicitno **(1) otvoriti, (2) koristiti i (3) zatvoriti**.

Primjeri resursa koji ovakvo zahtijevaju upravljanje uključuju:

- **datoteke** (otvaranje → čitanje/pisanje → zatvaranje)
- **mrežne veze** (otvaranje → slanje zahtjeva → zatvaranje)
- **baze podataka** (otvaranje → izvršavanje upita → zatvaranje)

Naredba `with` omogućava automatsko upravljanje resursima, osiguravajući da će se resursi pravilno osloboditi i zatvoriti čak i ako dođe do greške u bloku koda. Na taj način, kod postaje čišći i sigurniji za izvođenje.

Konteksti menadžer `with` je *blocking I/O* mehanizam, što znači da će se kod unutar `with` bloka izvršavati sekvencijalno, čekajući da se svaka operacija završi prije nego što priđe na sljedeću. *Non-blocking* varijanta kontekstnog menadžera koristi se u asinkronom programiranju, gdje se koristi `async with` sintaksa.

Sintaksa:

```
with neki_resurs as alias:  
    # rad s resursom koristeći "alias"
```

Tipičan primjer korištenje naredbe `with` je rad s datotekama:

```
with open("datoteka.txt", "r") as file: # otvaramo datoteku za čitanje i koristimo alias  
    "file"  
    sadržaj = file.read() # čitamo sadržaj datoteke  
    print(sadržaj)
```

Bez korištenja naredbe `with`, morali bismo eksplisitno zatvoriti datoteku nakon što smo pročitali sadržaj:

```
file = open("datoteka.txt", "r") # open je builtin funkcija za čitanje datoteka  
sadržaj = file.read()  
print(sadržaj)  
file.close() # zatvaramo datoteku
```

Međutim kod iznad ne obuhvaća slučaj greške prilikom čitanja ili pisanja u datoteku ako postoji. U tom slučaju, trebali bismo koristiti `try-except-finally` blokove kako bismo osigurali da će se datoteka zatvoriti čak i ako dođe do greške.

```
try:  
    file = open("datoteka.txt", "r")  
    sadržaj = file.read()  
    print(sadržaj)  
except Exception as e:  
    print(f"Greška: {e}")  
finally:  
    file.close()
```

Osim spomenutih primjera resursa, možemo definirati i [vlastite kontekstne menadžere](#), međutim to nije predmet ove skripte.

Naredba `with` **automatski zatvara resurs čak i ako dođe do greške u bloku koda**, što je jedan od razloga zašto se preporučuje njeno korištenje.

Dodatno, vidimo da je kod s naredbom `with` **kraći i lakši za čitanje**.

Obrada grešaka (eng. error handling) u Pythonu

U Pythonu, **upravljanje greškama** (eng. *error handling*) omogućava programerima da unaprijed predvide i pravilno reagiraju na pogreške koje se mogu pojaviti tijekom izvođenja programa. Temeljni alat za kontrolu i obradu takvih situacija jesu blokovi `try`, `except` i `finally`.

Sintaksa:

```

try:
    # kod koji može izazvati grešku
except Exception as e: # Exception je bazna klasa za većine iznimki u Pythonu, dok je "e"
    # alijsa za instancu iznimke
    # kod za rukovanje greškom
finally:
    # kod koji se uvijek izvršava, bez obzira na grešku

```

- **try blok:** Sadrži kod koji može potencijalno izazvati grešku. Ako se greška dogodi, Python će prekinuti izvođenje koda unutar `try` bloka i prijeći na odgovarajući `except` blok.
- **except blok:** Definira kako će program reagirati na određene vrste grešaka. Možemo specificirati različite vrste iznimki koje želimo uhvatiti i obraditi. Potrebno je definirati barem jedan `except` blok kako bismo uhvatili greške.
- **finally blok:** Sadrži kod koji će se uvijek izvršiti, bez obzira na to je li došlo do greške ili ne. Ovaj blok se često koristi za čišćenje resursa, poput zatvaranja datoteka ili mrežnih veza.

`Exception` je bazna klasa za većinu (ali ne baš sve) iznimka u Pythonu. Kada koristimo `except Exception as e`, uhvatit ćemo sve vrste iznimki koje se nasljeđuju iz klase `Exception`. Varijabla `e` predstavlja **instancu iznimke koja sadrži informacije o grešci**, poput poruke o grešci i stoga je korisna za dijagnostiku.

Česti primjeri iznimki koje možemo uhvatiti uključuju:

- `ValueError`: nastaje kada funkcija primi argument neprikladnog tipa ili vrijednosti.
- `TypeError`: nastaje kada se operacija ili funkcija primjeni na objekt neprikladnog tipa.
- `FileNotFoundException`: nastaje kada se pokuša pristupiti datoteci koja ne postoji.
- `ZeroDivisionError`: nastaje kada se pokuša dijeliti s nulom.
- `KeyError`: nastaje kada se pokuša pristupiti nepostojećem ključu u rječniku.
- `IndexError`: nastaje kada se pokuša pristupiti indeksu izvan raspona liste ili niza.
- `ConnectionError`: nastaje kada dođe do problema s mrežnom vezom.
- `TimeoutError`: nastaje kada operacija traje duže nego što je dopušteno vrijeme čekanja.
- `ImportError`: nastaje kada se pokuša uvesti modul koji ne postoji ili nije dostupan.

Primjer: Iznimku `ValueError` možemo uhvatiti na sljedeći način:

```

try:
    broj = int(input("Unesite cijeli broj: "))
except ValueError as e:
    print(f"Greška: Uneseni podatak nije cijeli broj. Detalji: {e}")

```

Primjer: Iznimku `FileNotFoundException` možemo uhvatiti na sljedeći način:

```
try: # nakon try radimo indentaciju
    with open("nepostojeca_datoteka.txt", "r") as file: # nakon with bloka također radimo
        indentaciju
            sadržaj = file.read()
except FileNotFoundError as e:
    print(f"Greška: Datoteka nije pronađena. Detalji: {e}")
```

2.2.2 ClientSession klasa

Vratimo se na naš primjer slanja 15 zahtjeva na *Cat Facts API*. Što je ovdje resurs koji trebamo otvoriti i zatvoriti, u kontekstu `with` naredbe?

► Spoiler alert! Odgovor na pitanje

U `aiohttp` biblioteci, za rad s HTTP sesijom koristimo klasu `ClientSession`.

Klasa `clientSession` predstavlja asinkroni HTTP klijent koji omogućuje **konkurentno slanje HTTP zahtjeva unutar Python programa**. Ovaj klijent implementiran je kao kontekstni menadžer, što znači da ga možemo koristiti unutar `with` bloka.

Kako bismo stvorili novu instancu `clientSession` klase, kao i klase općenito, jednostavno pozivamo njen konstruktor:

U varijablu `session` spremamo instancu klase `ClientSession`:

```
session = aiohttp.ClientSession()
```

Nakon što smo stvorili instancu klase, možemo koristiti `with` blok kako bismo definirali blok koda za asinkroni rad s HTTP sesijom. Jedina razlika je što sad stvari radimo asinkrono pa moramo koristiti `async` ispred kontekstnog menadžera.

Obzirom da koristimo `with`, možemo definirati alias `session` za **instancu klase unutar `async with` bloka koda**:

```
async with aiohttp.ClientSession() as session: # otvaramo asinkronu HTTP sesiju
    # rad s HTTP sesijom
```

Nad našom instancom sesije `session` sad možemo koristiti metodu `get` za slanje GET zahtjeva na isti način kao što smo to radili s *blocking requests* bibliotekom:

```
async with aiohttp.ClientSession() as session:
    response = await session.get("https://catfact.ninja/fact") # šalje HTTP GET zahtjeva na
    navedeni URL
    print(response)
```

Kako ovo sad pozvati? **Context manager sam po sebi nije funkcija, niti korutina**. Zato ćemo ga pozvati unutar korutine.

Prebacujemo cijeli kod unutar `main` korutine:

```

async def main(): # definiramo main korutinu
    async with aiohttp.ClientSession() as session: # otvaramo HTTP sesiju koristeći context
        manager "with"
        response = await session.get("https://catfact.ninja/fact") # mrežni zahtjev
        print(response)

# pokrećemo main korutinu i event loop koristeći asyncio.run() funkcije
asyncio.run(main())

```

Ako pokrenete kod vidjet ćete ogroman ispis, to je zato što smo ispisali **cijeli HTTP odgovor**, uključujući zaglavla, statusnu liniju, tijelo itd...

Kako bismo dobili samo tijelo odgovora, možemo na isti način kao i kod `requests` biblioteke koristiti metodu `json()` za deserijalizaciju, ali s jednom razlikom - moramo koristiti `await` ključnu riječ jer je metoda sada asinkrona, tj. **vraća objekt korutine**.

```

async def main():
    async with aiohttp.ClientSession() as session:
        response = await session.get("https://catfact.ninja/fact") # mrežni zahtjev
        fact_dict = await response.json() # dodajemo await jer je json() asinkrona metoda koja
        vraća objekt korutine
        print(fact_dict) # ispisuje nasumičnu činjenicu

```

OK, sada znamo kako poslati jedan zahtjev asinkrono. Vidimo da se trajanje nije promijenilo, ali to je zato što smo poslali samo jedan zahtjev i nismo izvršili konkurentnu obradu.

Idemo poslati 5 zahtjeva na ovaj način, jednostavno ćemo kod iterirati 5 puta.

```

async def main():
    async with aiohttp.ClientSession() as session:
        for i in range(5):
            response = await session.get("https://catfact.ninja/fact") # mrežni zahtjev
            fact_dict = await response.json() # deserijalizacija JSON odgovora
            print(fact_dict)

```

Trebali biste uočiti da stvari rade nešto brže nego prije, ali i dalje šaljemo zahtjeve sekvencijalno, čekajući odgovor prethodnog prije nego pošaljemo novi - svaki zahtjev šalje se u zasebnoj iteraciji `for` petlje.

Konkurentnost nismo postigli budući da **čekamo odgovor svakog zahtjeva prije nego pošaljemo sljedeći**, odnosno nismo rasporedili korutine za slanje zahtjeva u *event loop* - već izvršavamo *schedule and run* aktivnosti sekvencijalno - za svaku korutinu se čeka odgovor prije nego se rasporedi nova korutina u *event loop*.

2.2.3 Konkurentna obrada HTTP zahtjeva (`asyncio.gather`)

U prošloj skripti ste naučili da možemo koristiti `asyncio.gather` funkciju kako bismo **pozvali više korutina konkurentno i zatim pohraniti sve rezultate u jednu listu**, ili možemo koristiti `asyncio.create_task wrapper` kako bismo **stvorili zadatke Task objekte za svaku korutinu i dodali ih u event loop** prije nego ih pokrenemo.

Kako smo rekli da razmišljamo u kontekstu egzekucije korutina, idemo pokušati "izvući" kod za slanje zahtjeva u zasebnu korutinu (izvan `main` korutine):

Ideja je da iz sljedeće `main` korutine izvučemo kod za slanje zahtjeva u zasebnu korutinu `get_cat_fact`, budući da želimo spakirati ponavljajući kod u zasebnu korutinu:

```
async def main():
    async with aiohttp.ClientSession() as session:
        for i in range(5):
            print(f"Šaljemo {i + 1}. zahtjev...")
            response = await session.get("https://catfact.ninja/fact")
            fact_dict = await response.json()
            print(fact_dict['fact'])

asyncio.run(main())
```

Glavno pitanje je **gdje ćemo definirati context manager?** Unutar `main` korutine ili unutar `get_cat_fact` korutine?

► Spoiler alert! Odgovor na pitanje

U korutinu `get_cat_fact` proslijedujemo alias `session` kao njezin parametar:

```
async def get_cat_fact(session):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    return fact_dict
```

U `main` korutini tada moramo definirati otvaranje same sesije:

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

Napokon, možemo koristiti `asyncio.gather` funkciju kako bismo poslali 5 zahtjeva konkurentno.

- kako već znamo dobro *comprehension* sintaksu, iskoristit ćemo *list comprehension* za izradu liste korutina:

```

async def get_cat_fact(session):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(fact_dict['fact'])

async def main():
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]

```

Pozivamo korutine konkurentno koristeći `asyncio.gather` funkciju

```

async def main():
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]
    await asyncio.gather(*cat_fact_korutine)

```

Pokrenite kod - vidimo da se činjenice ispisuju dosta brzo.

A kitten will typically weigh about 3 ounces at birth. The typical male housecat will weigh between 7 and 9 pounds, slightly less for female housecats.
Cats see six times better in the dark and at night than humans.
There are approximately 60,000 hairs per square inch on the back of a cat and about 120,000 per square inch on its underside.
Cats bury their feces to cover their trails from predators.
The Egyptian Mau is probably the oldest breed of cat. In fact, the breed is so ancient that its name is the Egyptian word for "cat."

Ako se prisjetite, prosječno vrijeme trajanja programa s 5 činjenica je bilo 1-2 sekunde, ali tada smo imali i ispisivanje: `print("Šaljemo n. zahtjev...")` u svakoj iteraciji.

Dodat ćemo i ovdje `print` naredbu prije ispisa činjenice i izmjeriti vrijeme koristeći `time` modul:

```

async def get_cat_fact(session):
    print("Šaljemo zahtjev za mačji fact")
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(fact_dict['fact'])

```

I bez dodavanja `time` modula, odmah vidimo razliku u terminalu! Prije smo imali **sekvensijalno slanje zahtjeva po zahtjev i čekanje na odgovor prije slanja sljedećeg zahtjeva**:

Sinkrono slanje HTTP zahtjeva (`requests`):

Šaljemo 1. zahtjev...

Cats often overreact to unexpected stimuli because of their extremely sensitive nervous system.

Šaljemo 2. zahtjev...

The normal body temperature of a cat is between 100.5 ° and 102.5 °F. A cat is sick if its temperature goes below 100 ° or above 103 °F.

Šaljemo 3. zahtjev...

If they have ample water, cats can tolerate temperatures up to 133 °F.

Šaljemo 4. zahtjev...

Cats don't have sweat glands over their bodies like humans do. Instead, they sweat only through their paws.

Šaljemo 5. zahtjev...

The first commercially cloned pet was a cat named "Little Nicky." He cost his owner \$50,000, making him one of the most expensive cats ever.

Izvršavanje programa traje 1.26 sekundi.

Sada vidimo da se svi zahtjevi (korutine koje su *schedulane* u *event loopu*) prvo pošalju **konkurentno**, a zatim ispisuju sve činjenice. **Ne čekamo više na odgovor kroz svaku iteraciju petlje.**

Konkurentno slanje (*aiohttp*):

Šaljemo zahtjev za mačji fact

--- nakon toga odgovori se kreću ispisivati nasumičnim redoslijedom, ovisno o duljini trajanja blokirajuće mrežne operacije

Lions are the only cats that live in groups, called prides. Every female within the pride is usually related.

A happy cat holds her tail high and steady.

The average cat food meal is the equivalent to about five mice.

The Egyptian Mau is probably the oldest breed of cat. In fact, the breed is so ancient that its name is the Egyptian word for "cat."

A cat's nose pad is ridged with a unique pattern, just like the fingerprint of a human.

Zanima nas još i vrijeme izvođenja programa.

Započeti ćemo mjeriti kad se pozove `main` korutina, a završiti na kraju `main` korutine.

```

async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]
        await asyncio.gather(*cat_fact_korutine)
    end = time.perf_counter()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")

asyncio.run(main())

```

Primjer ispisa:

```

Šaljemo zahtjev za mačji fact
Cats have "nine lives" thanks to a flexible spine and powerful leg and back muscles
Cats' eyes shine in the dark because of the tapetum, a reflective layer in the eye, which
acts like a mirror.
The oldest cat on record was Crème Puff from Austin, Texas, who lived from 1967 to August
6, 2005, three days after her 38th birthday. A cat typically can live up to 20 years,
which is equivalent to about 96 human years.
When a cat rubs up against you, the cat is marking you with its scent claiming
ownership.
Cats see six times better in the dark and at night than humans.

Izvršavanje programa traje 0.27 sekundi.

```

Vidimo da se vrijeme izvršavanja programa na ovom jednostavnom primjeru slanja 5 zahtjeva **smanjilo s ~1.26 sekundi na ~0.27 sekundi.**

Razliku možemo izraziti i u postocima:

$$\frac{sekvencijalnoVrijeme - konkurentnoVrijeme}{sekvencijalnoVrijeme} \times 100 \quad (1)$$

odnosno:

$$\frac{1.26 - 0.27}{1.26} \times 100 \approx 78.57\% \quad (2)$$

Dakle, **konkurentni kod se izvršio otprilike 78.57% brže od sinkronog!**

Naravno, ovi izračuni ovise o mrežnim uvjetima i opterećenju poslužitelja, ali i brzini CPU-a gdje se konkurentna obrada odvija. Ipak, gotovo uvijek možemo očekivati značajno poboljšanje vremena izvođenja jer bolje iskorištavamo CPU vrijeme.

Ako podijelimo staro vrijeme izvršavanja s novim, vidimo da je **konkurentni kod gotovo 5 puta brži od sinkronog.**

$$\frac{sekvensijalnoVrijeme}{konkurentnoVrijeme} \quad (3)$$

$$\frac{1.26}{0.27} = 4.67 \quad (4)$$

Pokušajmo i s 15 zahtjeva:

```
async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(15)]
        await asyncio.gather(*cat_fact_korutine)
    end = time.perf_counter()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Primjer ispisa:

```
Šaljemo zahtjev za mačji fact
Cat families usually play best in even numbers. Cats and kittens should be acquired in pairs whenever possible.
Cats are subject to gum disease and to dental caries. They should have their teeth cleaned by the vet or the cat dentist once a year.
The biggest wildcat today is the Siberian Tiger. It can be more than 12 feet (3.6 m) long (about the size of a small car) and weigh up to 700 pounds (317 kg).
A group of cats is called a clowder.
The heaviest cat on record is Himmy, a Tabby from Queensland, Australia. He weighed nearly 47 pounds (21 kg). He died at the age of 10.
A cat can jump up to five times its own height in a single bound.
A commemorative tower was built in Scotland for a cat named Towser, who caught nearly 30,000 mice in her lifetime.
Purring does not always indicate that a cat is happy and healthy - some cats will purr loudly when they are terrified or in pain.
Long, muscular hind legs enable snow leopards to leap seven times their own body length in a single bound.
The most traveled cat is Hamlet, who escaped from his carrier while on a flight. He hid for seven weeks behind a panel on the airplane. By the time he was discovered, he had traveled nearly 373,000 miles (600,000 km).
Cats and kittens should be acquired in pairs whenever possible as cat families interact best in pairs.
```

The earliest ancestor of the modern cat lived about 30 million years ago. Scientists called it the Proailurus, which means "first cat" in Greek. The group of animals that pet cats belong to emerged around 12 million years ago.

There are up to 60 million feral cats in the United States alone.

The strongest climber among the big cats, a leopard can carry prey twice its weight up a tree.

The name "jaguar" comes from a Native American word meaning "he who kills with one leap".

Izvršavanje programa traje 0.61 sekundi.

Vidimo da se vrijeme izvršavanja programa s 15 zahtjeva **smanjilo s 3-4 sekunde na 0.61 sekundi**.

Ovdje nam je također program gotovo 5 puta brži, odnosno poboljšanje je ~80%.

2.2.4 Konkurentna obrada HTTP zahtjeva (`asyncio.Task`)

Naučili smo kako koristiti `asyncio.gather` funkciju za konkurentno izvođenje korutina. Međutim, u prošloj skripti smo rekli da možemo definirati i tzv. **Taskove** koji predstavljaju eventualni rezultat izvršavanja korutina unutar *event loopa*.

Rekli smo da `Task` objekti, omogućuju bolju kontrolu nad izvršavanjem korutina jer možemo pratiti njihov status, upravljati njima pojedinačno, i eventualno čekati pojedinačne rezultate korutina pomoći `await` ključne riječi, za razliku od `asyncio.gather` funkcije koja nam vraća sve rezultate odjednom.

U našem primjeru dohvaćanja činjenica o mačkama, korutine su `get_cat_fact`. Možemo ih jednostavno pohraniti u listu i zatim izraditi `Task` objekte za svaku, koristeći *list comprehension*.

Nakon toga ćemo ih pozvati koristeći `await` ključnu riječ jednostavnim iteriranjem kroz listu `Task` objekata.

```
async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] # pohranujemo Taskove u listu korutina i RASPOREĐUJEMO ih u event loopu
        for task in cat_fact_tasks: # ovaj kod izvršava se konkurentno jer smo koristili Taskove
            await task # čekamo rezultate svakog taska, dok će prvi await (cat_fact_tasks[0]) pokrenuti sve korutine koje smo rasporedili u event loopu
    end = time.perf_counter()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Rezultat je identičan kao i kod `asyncio.gather` funkcije:

Šaljemo zahtjev za mačji fact

70% of your cat's life is spent asleep.

Cats eat grass to aid their digestion and to help them get rid of any fur in their stomachs.

In 1987 cats overtook dogs as the number one pet in America.

In ancient Egypt, when a family cat died, all family members would shave their eyebrows as a sign of mourning.

A cat can't climb head first down a tree because every claw on a cat's paw points the same way. To get down from a tree, a cat must back down.

Izvršavanje programa traje 0.28 sekundi.

Ako izvrtimo kod više puta, vidjet ćete da je rezultat izvođenja identičan (~0,27 sekundi) kao što je to bio slučaj s `asyncio.gather` funkcijom.

Rekli smo da je moguće i kombinirati ova dva pristupa, odnosno koristiti `asyncio.gather` funkciju za konkurentno izvođenje `Taskova`, međutim ovo je pomalo redundantno budući da `gather` automatski *wrappa* korutine u `Task` objekte.

```
async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] #
    pohranujemo Taskove u listu
        await asyncio.gather(*cat_fact_tasks) # pozivamo Taskove konkurentno
    end = time.perf_counter()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Možemo maknuti `print` naredbe unutar korutine `get_cat_fact` te vratiti samo činjenicu kao rezultat te korutine:

```
async def get_cat_fact(session):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    return fact_dict['fact']

async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] #
    pohranujemo Taskove u listu
        actual_cat_facts = await asyncio.gather(*cat_fact_tasks) # pohranit ćemo rezultate u
    listu
    end = time.perf_counter()
    print(actual_cat_facts)
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")

asyncio.run(main())
```

Rezultat je lista činjenica:

```
[ 'The Maine Coone is the only native American long haired breed.', 'The Amur leopard is
one of the most endangered animals in the world.', "A cat's normal pulse is 140-240 beats
per minute, with an average of 195.", 'The cat has 500 skeletal muscles (humans have
650).', 'A happy cat holds her tail high and steady.]
```

Izvršavanje programa traje 0.27 sekundi.

U slučaju da nam ispisi i vrijeme izvođenja nisu dovoljan dokaz da su zahtjevi uistinu poslani konkurentno, možemo još provjeriti **redoslijed ispisivanja činjenica** koji nam je, ako se prisjetite, kod sekvencijalnog slanja uvijek bio isti: 1 2 3 4 5.

Ovdje to možemo testirati na način da ćemo jednostavno proslijediti `i` lokalnu varijablu iz petlje u korutinu `get_cat_fact`:

```
async def get_cat_fact(session, i):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(f"{i + 1}: {fact_dict['fact']}") # dodajemo ispis u formatu: "redniBroj:
    #činjenica"
    return fact_dict['fact']

async def main():
    start = time.perf_counter()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session, i)) for i in range(5)] # u
        #korutinu proslijedujemo parametar "session" i lokalnu varijablu "i"
        actual_cat_facts = await asyncio.gather(*cat_fact_tasks)
    end = time.perf_counter()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
asyncio.run(main())
```

Primjer ispisa (round 1):

```
2: It is estimated that cats can make over 60 different sounds.
1: According to a Gallup poll, most American pet owners obtain their cats by adopting
strays.
5: Cats are the world's most popular pets, outnumbering dogs by as many as three to one
3: The oldest cat to give birth was Kitty who, at the age of 30, gave birth to two
kittens. During her life, she gave birth to 218 kittens.
4: Cats can jump up to 7 times their tail length.
```

Primjer ispisa (round 2):

```
1: In Japan, cats are thought to have the power to turn into super spirits when they die.
This may be because according to the Buddhist religion, the body of the cat is the
temporary resting place of very spiritual people.i
4: A cat sees about 6 times better than a human at night, and needs 1/6 the amount of light
that a human does - it has a layer of extra reflecting cells which absorb light.
3: Cats lived with soldiers in trenches, where they killed mice during World War I.
5: In relation to their body size, cats have the largest eyes of any mammal.
2: Female felines are \superfecund
```

Primjer ispisa (round 3):

- 4: Mountain lions are strong jumpers, thanks to muscular hind legs that are longer than their front legs.
- 2: Cats' hearing stops at 65 khz (kilohertz); humans' hearing stops at 20 khz.
- 3: Retractable claws are a physical phenomenon that sets cats apart from the rest of the animal kingdom. In the cat family, only cheetahs cannot retract their claws.
- 5: A cat uses its whiskers for measuring distances. The whiskers of a cat are capable of registering very small changes in air pressure.
- 1: Tylenol and chocolate are both poisionous to cats.

Ako se prisjetimo ilustracije konkurentnog izvođenja na samom početku skripte RS3, da se zaključiti zašto su rezultati ovakvi.

► Spoiler alert! Odgovor na pitanje

Sada definitivno možemo reći da je kod koji smo definirali **konkurentan** te možemo uočiti **konkretna poboljšanja u brzini izvođenja programa** 

2.3 Timeout mrežnih operacija i obrada iznimki

Kada radimo s mrežnim operacijama, **uvijek postoji mogućnost da dođe do kašnjenja ili prekida veze**. Zbog toga je važno postaviti vremenska ograničenja (*timeout*) za naše mrežne zahtjeve kako bismo spriječili da naš program "visi" čekajući odgovor koji možda nikada neće stići.

Zadano ponašanje `asyncio.gather` funkcije je da čeka na sve korutine da se završe, bez vremenskog ograničenja. Ovo može biti praktično, ali i opasno ako neka od korutina traje predugo zbog mrežnih problema.

Primjer: Naš raspodijeljeni sustav za praćenje prometa motornih vozila šalje zahtjeve na različite *monitoring* kamere na cestama koje su instalirane na udaljenim lokacijama. Postoje mikroservisi za svaku kameru koji obrađuju slike i imaju REST API za dohvaćanje podataka kako bi komunicirali s glavnim sustavom. Zamislite da je glavni sustav ustvari naša `main` korutina koja šalje zahtjeve na različite kamere koristeći `asyncio.gather` funkciju, gdje svaki zahtjev predstavlja korutinu koja šalje HTTP zahtjev na različite API-je kamera.

Neke kamere su udaljenije, na nepovolnjim geografskim lokacijama, te je veća vjerojatnost da će doći do mrežnih problema ili kašnjenja.

Kako mi ne možemo pravilno obraditi podatke bez svih odgovora (podataka svih kamera), a opet ne želimo da naš glavni sustav "visi" čekajući odgovore koji možda nikada neće stići, možemo postaviti vremensko ograničenje za svaki zahtjev.

To možemo postići korištenjem `asyncio.wait_for` funkcije koja omogućuje postavljanje vremenskog ograničenja za izvršavanje korutine. Ako korutina ne završi unutar zadanoog vremena, `asyncio.TimeoutError` iznimka će biti podignuta.

Sintaksa:

```
await asyncio.wait_for(coroutine, timeout)
```

- `coroutine`: korutina koju želimo izvršiti s vremenskim ograničenjem
- `timeout`: maksimalno vrijeme (u sekundama) koje korutina smije trajati

Iznimke u Pythonu možemo podići (simulirati) i sami, naredbom `raise`:

Sintaksa:

```
raise ExceptionType("poruka o grešci")
```

- gdje `ExceptionType` predstavlja tip iznimke koju želimo podići (npr. `ValueError`, `TimeoutError`, itd.)

Definirat ćemo nekoliko korutina koje simuliraju mrežne operacije naših mikroservisa za praćenje prometa motornih vozila.

```
import asyncio

async def fetch_camera_data(camera_id):
```

```

print(f"Fetching data from camera {camera_id}...")
# Simuliramo mrežnu operaciju s različitim vremenima trajanja
await asyncio.sleep(camera_id * 2) # Kamera s ID-jem 1 traje 2 sekunde, ID-jem 2
traje 4 sekunde, itd.
if camera_id == 3:
    raise Exception("Camera 3 is unreachable!") # Simuliramo grešku za kameru 3
(kamera koja se nalazi na nepovoljnoj lokaciji)
print(f"Data from camera {camera_id} fetched.")
return f"Data from camera {camera_id}"

async def main():
    camera_ids = [1, 2, 3, 4, 5]
    tasks = [fetch_camera_data(camera_id) for camera_id in camera_ids]

    # Primjer asyncio.gather bez timeouta

    try:
        results = await asyncio.gather(*tasks)
        print("All camera data fetched:", results)
    except Exception as e:
        print("An error occurred while fetching camera data:", e)

asyncio.run(main())

```

Ispisuje:

```

Fetching data from camera 1...
Fetching data from camera 2...
Fetching data from camera 3...
Fetching data from camera 4...
Fetching data from camera 5...
Data from camera 1 fetched.
Data from camera 2 fetched.
An error occurred while fetching camera data: Camera 3 is unreachable!

```

Možemo simulirati situaciju da zahtjev kamere 3 traje predugo, npr. 20 sekundi, ali će se ipak naposljeku podatak nastaviti:

```

async def fetch_camera_data(camera_id):
    print(f"Fetching data from camera {camera_id}...")
    # Simuliramo mrežnu operaciju s različitim vremenima trajanja
    await asyncio.sleep(camera_id * 2 if camera_id != 3 else 20) # kamera 3 traje 20
sekundi, sve ostale vraćaju odgovor nakon 2 sekunde
    print(f"Data from camera {camera_id} fetched.")
    return f"Data from camera {camera_id}"

```

Ispisuje:

```
Fetching data from camera 1...
Fetching data from camera 2...
Fetching data from camera 3...
Fetching data from camera 4...
Fetching data from camera 5...
Data from camera 1 fetched.
Data from camera 2 fetched.
Data from camera 4 fetched.
Data from camera 5 fetched.
Data from camera 3 fetched.
All camera data fetched: ['Data from camera 1', 'Data from camera 2', 'Data from camera 3', 'Data from camera 4', 'Data from camera 5']
```

Uočite da se rezultati kamere 4 i 5 ispisuju prije rezultata kamere 3, budući da je kamera 3 simulirana kao najsporija, a zahtjevi su poslani konkurentno.

Ipak, želimo postaviti vremensko ograničenje od 10 sekundi za svaki zahtjev, kako `asyncio.gather` ne bi bilo usko grlo našeg programa koje čeka do 20 sekundi na odgovor kamere 3.

Koristimo funkciju `asyncio.wait_for`. Ova funkcija očekuje bilo koju korutinu kao ulazni podatak, i vrijeme maksimalnog čekanja na izvršavanje u sekundama.

Sintaksa:

```
await asyncio.wait_for(coroutine, timeout)
```

- gdje je `coroutine` korutina koju želimo izvršiti s vremenskim ograničenjem, a `timeout` maksimalno vrijeme (u sekundama) koje korutina smije trajati.

Možemo ograničiti `asyncio.gather()` s `asyncio.wait_for()` na sljedeći način:

```
async def main():
    camera_ids = [1, 2, 3, 4, 5]
    tasks = [fetch_camera_data(camera_id) for camera_id in camera_ids]

    try:
        results = await asyncio.wait_for(asyncio.gather(*tasks), timeout=10) # Čekaj na sve korutine maksimalno 10 sekundi
        print("All camera data fetched:", results)
    except asyncio.TimeoutError:
        print("A timeout occurred while fetching camera data.")
    except Exception as e:
        print("An error occurred while fetching camera data:", e)
```

Ispisuje:

```
Fetching data from camera 1...
Fetching data from camera 2...
Fetching data from camera 3...
Fetching data from camera 4...
Fetching data from camera 5...
Data from camera 1 fetched.
Data from camera 2 fetched.
Data from camera 4 fetched.
A timeout occurred while fetching camera data.
```

Uočite da smo dobili `TimeoutError` te se aktivirao `except` blok za hvatanje iznimke.

Ipak, možemo ograničiti i dohvaćanje podataka za svaku kameru pojedinačno, budući da je `asyncio.sleep` također korutina.

```
async def fetch_camera_data(camera_id):
    print(f"Fetching data from camera {camera_id}...")
    try:
        await asyncio.wait_for(asyncio.sleep(camera_id * 2 if camera_id != 3 else 20),
                           timeout=10) # Postavljamo timeout od 10 sekundi za dohvaćanje podataka o svakoj kameri
                           pojedinačno
        print(f"Data from camera {camera_id} fetched.")
        return f"Data from camera {camera_id}"
    except asyncio.TimeoutError:
        print(f"Timeout while fetching data from camera {camera_id}.")
        return None
```

Nakon toga možemo koristiti `asyncio.gather` kao i prije:

```
async def main():
    camera_ids = [1, 2, 3, 4, 5]
    tasks = [fetch_camera_data(camera_id) for camera_id in camera_ids]
    results = await asyncio.gather(*tasks)
    print("All camera data fetched:", results)
```

`asyncio.gather` će sada dohvatiti podatke iz svih korutina, a one koje su premašile vremensko ograničenje vratit će `None`.

Ispisuje:

```
Fetching data from camera 1...
Fetching data from camera 2...
Fetching data from camera 3...
Fetching data from camera 4...
Fetching data from camera 5...
Data from camera 1 fetched.
Data from camera 2 fetched.
Data from camera 4 fetched.
Timeout while fetching data from camera 3.
Timeout while fetching data from camera 5.
All camera data fetched: [ 'Data from camera 1', 'Data from camera 2', None, 'Data from
camera 4', None]
```

Više o rukovanju iznimkama i timeout-ima radit ćemo na budućim vježbama.

3. Zadaci za vježbu - Konkurentna obrada mrežnih operacija i simulacije grešaka

Zadatak 1: fetch_users i izdvajanje podataka

Definirajte korutinu `fetch_users` koja će slati GET zahtjev na [JSONPlaceholder API](#) na URL-u:

`https://jsonplaceholder.typicode.com/users`. Morate simulirati slanje 5 zahtjeva konkurentno unutar `main` korutine. Unutar `main` korutine izmjerite vrijeme izvođenja programa, a rezultate pohranite u listu odjedanput koristeći `asyncio.gather` funkciju. Nakon toga koristeći `map` funkcije ili *list comprehension* izdvojite u zasebne 3 liste: samo **imena korisnika**, samo **email adrese korisnika** i samo **username korisnika**. Na kraju `main` korutine ispišite sve 3 liste i vrijeme izvođenja programa.

Zadatak 2: filter_cat_facts

Definirajte dvije korutine, od kojih će jedna služiti za dohvaćanje činjenica o mačkama koristeći `get_cat_fact` korutinu koja šalje GET zahtjev na URL: `https://catfact.ninja/fact`. Izradite 20 `Task` objekata za dohvaćanje činjenica o mačkama te ih pozovite unutar `main` korutine i rezultate pohranite odjednom koristeći `asyncio.gather` funkciju. Druga korutina `filter_cat_facts` ne šalje HTTP zahtjeve, već zaprima **gotovu listu činjenica (stringova) o mačkama** i vraća novu listu koja sadrži samo one činjenice koje sadrže riječ "cat" ili "cats" (neovisno o velikim/malim slovima).

Primjer konačnog ispisa:

Filtrirane činjenice o mačkama:

- A 2007 Gallup poll revealed that both men and women were equally likely to own a cat.
- The first cat in space was a French cat named Felicette (a.k.a. "Astrocat") In 1963, France blasted the cat into outer space. Electrodes implanted in her brains sent neurological signals back to Earth. She survived the trip.
- The lightest cat on record is a blue point Himalayan called Tinker Toy, who weighed 1 pound, 6 ounces (616 g). Tinker Toy was 2.75 inches (7 cm) tall and 7.5 inches (19 cm) long.
- The first commercially cloned pet was a cat named "Little Nicky." He cost his owner \$50,000, making him one of the most expensive cats ever.
- In the 1750s, Europeans introduced cats into the Americas to control pests.
- A group of cats is called a clowder.

Zadatak 3: mix_dog_cat_facts

Definirajte korutinu `get_dog_fact` koja dohvaća činjenice o psima sa [DOG API](#) servisa.

Korutina `get_dog_fact` neka dohvaća činjenicu o psima na URL-u: <https://dogapi.dog/api/v2/facts>. Nakon toga, definirajte korutinu `get_cat_fact` koja dohvaća činjenicu o mačkama slanjem zahtjeva na URL: <https://catfact.ninja/fact>.

Istovremeno pohranite rezultate izvršavanja ovih Taskova koristeći `asyncio.gather(*dog_facts_tasks, *cat_facts_tasks)` funkciju u listu `dog_cat_facts`, a zatim ih koristeći *list slicing* odvojite u dvije liste obzirom da znate da je prvih 5 činjenica o psima, a drugih 5 o mačkama (bez obzira što mrežni rezultati različito "dolaze", gather ih pohranjuje redoslijedom poziva).

Na kraju definirajte treću korutinu `mix_facts` koja prima dvije liste, `dog_facts` i `cat_facts`, te vraća novu listu u kojoj se za svaki indeks `i` nalazi odabrana činjenica prema sljedećem pravilu: uzmite činjenicu o psima ako je njezina duljina veća od duljine odgovarajuće mačje činjenice; u suprotnom odaberite mačju činjenicu. Za paralelnu iteraciju dviju lista upotrijebite funkciju `zip`, npr. `for dog_fact, cat_fact in zip(dog_facts, cat_facts)`. Nakon dobivanja nove liste, ispišite filtrirani skup činjenica.

Primjer konačnog ispisa:

Mixane činjenice o psima i mačkama:

If they have ample water, cats can tolerate temperatures up to 133 °F.
Dogs with little human contact in the first three months typically don't make good pets.
The most popular dog breed in Canada, U.S., and Great Britain is the Labrador retriever.
An estimated 1,000,000 dogs in the U.S. have been named as the primary beneficiaries in their owner's will.
When a cat rubs up against you, the cat is marking you with its scent claiming ownership.

Zadatak 4: simulacija autentifikacije korisnika

Napišite korutinu `autentifikacija` koja simulira proces autentifikacije korisnika. Korutina treba primiti korisničko ime i lozinku, zatim simulirati sporo I/O čekanje (npr. 2 sekunde) prije nego što vrati `True` ako su korisničko ime i lozinka ispravni. Korisničko ime i lozinku provjerite prema rječniku `korisnici` koji sadrži parove korisničko ime-lozinka.

```

korisnici = {
    "korisnik1": "lozinka1",
    "korisnik2": "lozinka2",
    "korisnik3": "lozinka3",
}

```

Simulirajte pogrešku u autentifikaciji ako su uneseni podaci netočni (`raise ValueError`).

- Napišite glavnu funkciju koja će poslati konkurentne zahtjeve za autentifikaciju za 5 različitih korisnika (neki s ispravnim, neki s neispravnim podacima). Kako se ponaša `asyncio.gather()` kada se dogodi iznimka u jednoj od korutina?

Izmijenite kod korutine i simulirajte grešku u autentifikaciji koja se javlja **odmah** nakon 3 sekunde čekanja (npr. ne radi autentifikacijski servis) koji će podići iznimku `TimeoutError`.

- Dodajte `timeout` prilikom **poziva korutine** `autentifikacija` kako biste simulirali situaciju kada autentifikacijski servis ne odgovara na vrijeme.

Zadatak 5: Pretvorba sinkronog koda u asinkroni

Sljedeći isječak programskog koda pretvorite u asinkroni program s konkurentnom obradom mrežnih zahtjeva:**

```

import requests

def fetch_url(url: str) -> str:
    response = requests.get(url, timeout=5)
    return response.text

def main():
    urls = [
        "https://example.com",
        "https://httpbin.org/get",
        "https://api.github.com"
    ]

    for url in urls:
        content = fetch_url(url)
        print(f"Fetched {len(content)} characters from {url}")

if __name__ == "__main__":
    main()

```

Zadatak 6: Simulacija raspodijeljenog sustava za dohvaćanje i obradu vremenskih podataka

Radite na raspodijeljenom sustavu za dohvaćanje vremenskih podataka s različitih javnih API-ja**. Vaš servis treba konkurentno agregirati podatke o vremenu iz više izvora te nakon toga izračunati i ispisati prosječnu temperaturu. Definirajte korutinu `fetch_weather_data` (predstavlja mikroservis koji vraća podatke s meteorološke stanice na određenoj lokaciji), koja simulira određeno čekanje (možete staviti

nasumično čekanje između 1 i 5 sekundi koristeći `random.uniform(1, 5)` i vraća nasumičnu temperaturu između 20 i 25 stupnjeva Celzijusa. U glavnoj korutini `main` kreirajte i rasporedite 10 objekata tipa Task za konkurentno dohvaćanje vremenskih podataka s 10 različitih vremenskih stanica. Nakon što dobijete sve rezultate, izračunajte i ispišite prosječnu temperaturu.

- Simulirajte situaciju u kojoj nekoliko vremenskih stanica ne odgovara na vrijeme te pravilno obradite iznimku `TimeoutError`.
- Ograničite vrijeme čekanja na svaki zahtjev na najviše 2 sekunde; u suprotnom slučaju vratite `None` te izračunajte prosječnu temperaturu bez podataka za tu mjernu stanicu.

Ako hoćete, možete određene dijelove koda rasporediti u zasebne datoteke (module) ili možete sve napisati u jednoj datoteci.