Univerzitet u Beogradu
Matematički fakultet

Master rad

Razvoj asinhronih veb aplikacija korišćenjem radnog okvira Tornado

Autor:
Stefan Marić

Mentor:
doc. dr Milan Banković

Članovi komisije:

prof. dr Filip Marić, Matematički fakultet, Univerzitet u Beogradu

prof. dr Saša Malkov, Matematički fakultet, Univerzitet u Beogradu

Beograd, 2020

"Mi smo iskra u smrtnu prašinu, mi smo luča tamom obuzeta."

P.P. Njegoš

Sažetak

Pojam asinhronosti u programiranju odnosi se na postojanje događaja nezavisnih od glavnog toka programa, kao i na načine njihove obrade. Ovi događaji mogu biti raznorodni, no u polju razvoja veb aplikacija obično se razmatraju događaji čija obrada zahteva dosta vremena, poput obrade velike količine podataka iz baze, ili pristupa eksternim aplikativnim programskim interfejsima. Nekadašnji, sekvencijalni, pristup podrazumevao je zaustavljanje aplikacije do trenutka kada resursi ne postanu dostupni, nakon čega program može da nastavi sa radom. U današnjem vremenu, kada su veb aplikacije značajno kompleksnije i eksterni upiti su postali rutina, ovakav pristup nije zadovoljavajuć. Asinhroni pristup predstavlja jednu od tehnika poboljšanja i ogleda se u mogućnosti da se kontrola toka prepusti drugim delovima programa koji ne zavise od rezultata neke spore operacije, dok se ta operacija ne završi. U ovom radu prikazujemo tehnike i koncepte razvoja veb aplikacija u asinhronom maniru, u programskom jeziku *Python* i radnom okviru *Tornado*. Rad pruža kratak uvid u neke osnovne funkcionalnosti i mogućnosti programskog jezika *Python*, sa naglaskom na integraciji asinhronog modela u ovaj programski jezik. Rad takođe detaljno razmatra mogućnosti radnog okvira *Tornado*. Pored prikaza opšte strukture aplikacija pisanih u ovom okruženju, u radu je opisan i ugrađeni jezik veb šablona, razni bezbednosni mehanizmi, rad sa bazama podataka, kao i prodrška za asinhrono programiranje u *Tornado* okruženju. U radu je dat i opis veb aplikacije koja je razvijena za potrebe ove teze, a koja korisniku pruža prikaz vesti iz sveta prikupljene iz različitih izvora, ilustrujući pritom asinhroni pristup razvoja veb aplikacije, kao i mogućnosti radnog okvira *Tornado*.

Sadržaj

Poglavlje 1

Uvod i motivacija

1.1 Razvoj veba

Razvoj veba, odnosno razvoj softvera za veb (engl. *web development*) predstavlja skup različitih poslova koji se obavljaju prilikom razvoja veb-sajta (engl. *website*) ili veb-aplikacije (engl. *web application*). U te poslove ubrajamo veb-dizajn (engl. *web design*), veb programiranje, programiranje baza podataka, administraciju servera, upravljanje sadržajem, marketing i sl.

U cilju boljeg razumevanja čitavog procesa razvoja veb-aplikacija, uvešćemo podelu na razvoj klijentskog dela (engl. *front end*) i serverskog dela (engl. *back end*). Iako se radi o prilično različitim tehnikama i tehnologijama, za ispravno i neometano funkcionisanje jedne aplikacije, bitan je rad nad obema stranama.

Klijentski deo predstavlja deo koji se pokreće i izvršava u okviru klijentovog veb pregledača. Klijentu se prikazuje isporučeni *HTML* sadržaj, pri čemu se prikaz sadržaja stilizuje korišćenjem jezika *CSS*. Ukoliko postoje funkcionalnosti koje se izvršavaju na klijentskom delu (poput animacija, validacije formulara i drugih vidova interakcije sa korisnikom), za njih je zadužen programski jezik *JavaScript* (i tada već govorimo o dinamičkim veb stranama[1]). Od ostalih poslova izdvajaju se i *Ajax* zahtevi, optimizacija za pretraživače (engl. *search engine optimization*), kompatabilnost pregledača (engl. *cross-browser compatability*), prilagođavanje ekranu (engl. *responsiveness*), uređivanje slika itd.

Serverski deo je onaj deo veb-aplikacije koji se izvršava na serveru. Zadužen je za dinamičko generisanje sadržaja koji se isporučuje klijentu, kao i čuvanje i ažuriranje podataka koji se razmenjuju sa klijentom. Od dodatnih zaduženja posebno su bitni poslovi administracije baze podataka, arhitekture servera, poslovi bezbednosti, skalabilnosti, održavanje rezervne sigurnosne kopije (engl. *backup*) itd. Neki od popularnih skript jezika u ovom domenu su *JavaScript*, *ASP.NET*, *PHP*, *JSP*, *Ruby*, *Python*, ali i neki od kompiliranih programskih jezika poput *C#*, *Java* i *Go*.

[1] U današnjoj terminologiji često se koriste termini *aktivna strana* – ona čiji se sadržaj menja u pregledaču i *generisana strana* – ona koja se generiše na serveru. Shodno tome, na delu su sve četiri kombinacije: *neaktivna negenerisana* – statička, po staroj terminologiji, *aktivna negenerisana* – fiksirani *HTML* kod sa ugrađenim *JavaScript* kodom, kao i *neaktivna generisana* – dinamička, po staroj terminologiji i *aktivna generisana* strana, što predstavlja serverski generisan *HTML* kod koji se menja posredstvom *JavaScript*-a.

1.2 Asinhrono programiranje

Pojam asinhronosti u programiranju se odnosi na postojanje asinhronih *događaja* (tj. događaja koji se mogu desiti van glavnog toka izvršavanja), kao i mehanizama za njihovu obradu, nezavisnih od upravljanja tokom (engl. *control flow*). Takvi događaji mogu biti međuprocesni signali, ili akcije podstreknute od strane programa koje će se izvršiti konkurentno sa izvršavanjem glavnog programa, bez potrebe blokiranja zarad čekanja rezultata [4].

Asinhroni ulaz i izlaz je jedan primer takvog režima rada. Program je slobodan da izdaje komande ka bazi podataka ili ka mrežnim uređajima koji će uslužiti ove zahteve, dok procesor nastavlja sa daljim izvršavanjem programa. Ovakav pristup je naročito koristan u slučaju problema *vezanih za ulaz i izlaz* (engl. *I/O bound*), kod kojih je vreme izvršavanja u najvećoj meri određeno vremenom provedenim na čekanje ulaza ili izlaza (opet, u slučaju veba, to bi mogao biti upit nad bazom podataka, ili čekanje na rezultate nekog eksternog aplikativnog programskog interfejsa). U tom kontekstu, asinhronost nam može pomoći da "jeftino" čekamo, odnosno, da postoji napredak u izvršenju rada programa dok se ne završe poslovi ulaza ili izaza.

Ovaj slučaj adekvatno opisuje okolnosti pri razvoju veb-aplikacija i u daljem radu, fokus će biti na korišćenju asinhronosti za rešavanje ovakvih problema. Nekadašnji princip funkcionisanja veb aplikacija sastojao se od toga da klijent, korišćenjem svog veb pregledača, potražuje podatke od servera i *čeka* na njihovo dostavljanje pregledaču, koji će ih potom obraditi i sve to prikazati klijentu u jednoj čitljivoj formi. Iako je ovaj pristup i danas moguć, uključivanjem dodatnih funkcionalnosti i komplikovanjem izvršavanja procesa koji sadržaj generiše i prikazuje krajnjem korisniku, on često nije zadovoljajuć. U slučaju velikog broja konkurentnih zahteva, ovakav pristup bi mogao dovesti do privremenog "zamrzavanja" stranice korisnikovog pregledača, što negativno utiče na funkcionalnost i udobnost pri korišćenju veb aplikacije. Jedno od poboljšanja dolazi u formi asinhronog veb pristupa. Sa serverske strane, ovo znači da dok čekamo na podatke za jednog klijenta nećemo ignorisati zahteve drugih klijenata, već ćemo ih organizovati tako da procesor bude maksimalno upošljen, a vreme čekanja - minimalno. S druge, klijentske, strane ovo znači da klijentu omogućimo prikaz i odziv prilikom interakcije sa do sada već učitanim sadržajem, dok u pozadini dovlačimo i kreiramo nov.

Treba napomenuti da asinhronost nije lek za sve. Ukoliko u izvršavanju programa dominira izračunavanje, a ne ulaz i izlaz, tada će vreme izvršavanja u najvećoj meri biti određeno vremenom koje je potrebno procesoru da obradi datu količinu podataka. Za ovakve programe kazemo da su *vezani za procesor* (engl. *CPU bound*), i u takvim slučajevima asinhroni pristup nije od velike koristi. Dodatno, kod asinhronog pristupa, često je potreban dodatan napor da bi se obezbedila usklađenost klijentskog interfejsa sa podacima. Naime, često je potrebno blokirati, ili uskladiti, čitave delove interfejsa do stizanja podataka od servera.[2]

[2]Primer ovoga mogu biti veb stranice koje korisniku pružaju opcije filtriranja sadržaja. Ukoliko se, po odabranom filteru, izabere drugi pre stizanja samog sadržaja, može se desiti da se jedan od izbora "izgubi".

Kako bismo uverili čitaoca u svrsishodnost asinhronog pristupa u modernom računarstvu, ilustrujmo prethodno navedeno slikovitim primerom jednog konobara u restoranu. U sinhronom režimu, po ulasku gosta u restoran, konobar prilazi i uzima porudžbinu. Zatim odlazi do kuhinje, predaje porudžbinu kuvaru i čeka da bude gotova, ne mareći da li možda postoji još gostiju koji dolaze i spremni su da poruče hranu. Kada je poručeno - spremljeno, konobar odnosi hranu gostu i sada je spreman da ugosti sledećeg ko je ušao u restoran. I više je nego jasno da je, u slučaju velikog broja gostiju, poslovanje ovakvog (sinhronog) restorana daleko od zadovoljavajućeg. Poboljšanje ovog modela ogleda se u primeni asinhronog pristupa od strane konobara. Dakle, u asinhronom režimu, po primanju porudžbine, i predaji kuvaru, konobar odlazi do sledećeg gosta koji je spreman da poruči, njegovu porudžbinu odnosi u kuhinju. Pretpostavljajući da uvek ima gostiju spremnih da poruče, ovaj proces se ponavlja dokle god nečija porudžbina nije spremljena. Tada, naš asinhroni konobar odnosi jelo specifičnom gostu i po završetku, spreman je za primanje novih podrudžbina. Preselivši ovu alegoriju u domen veb programiranja, gosti bi bili korisnici, konobar – proces koji se izvršava, a kuvar i tok spremanja hrane bi verno opisivali poziv nekog spoljašnjeg interfejsa ili manipulaciju nad bazom podataka. Vredi napomenuti i ostale mogućnosti poboljšanja početnog pristupa. Jedna je povećavanje broja konobara koji bi opsluživali goste (što u kontekstu računara predstavlja uvođenje dodatnih niti), ili povećanje broja kuvara – interfejsa za dobavljanje podataka.

1.3 Python kao jezik veba

Programski jezik *Python* je trenutno jedan od najpopularnijih izbora programera (slika 1.1). Uz raznovrsne mogućnosti primene u različitim sferama, poput analize podataka, razvoja desktop aplikacija, administracije sistema, naučnih i numeričkih zadataka, on se ističe i u razvoju veb i internet tehnologija [2]. Neke od pogodnosti koje ovaj programski jezik omogućava pri razvoju veb aplikacija su:

Podrška za internet tehnologije i protokole:
Python-ova standardna biblioteka podržava mnoge internet tehnologije i protokole poput *HTML*, *XML*, *JSON*, e-mail procesiranje, podrška za *FTP*, *IMAP* i druge internet protokole, kao i podršku za rad sa veb utičnicama.

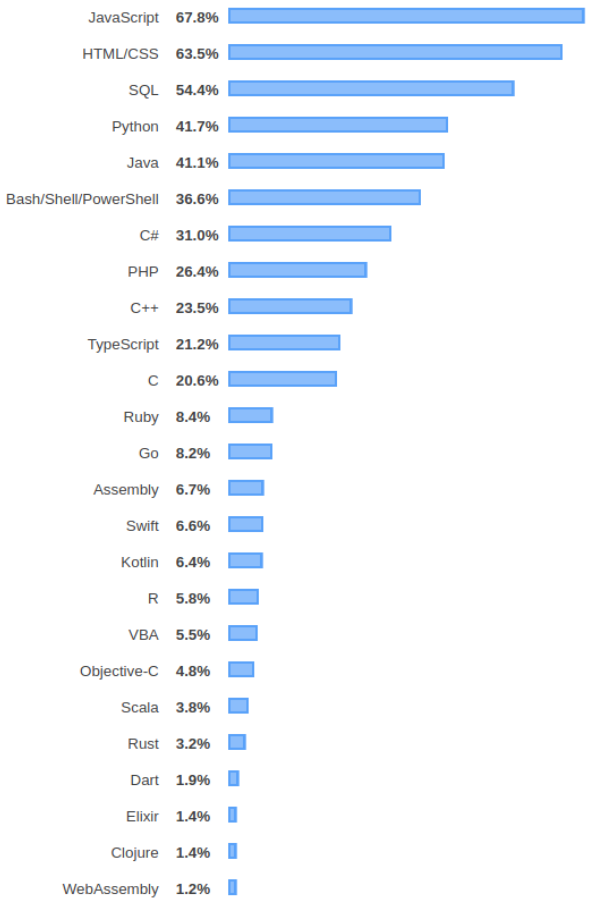
Opsežan repozitorijum gotovih paketa PyPI:
Ospežan repozitorijum gotovih paketa (engl. *Package Index*) (skr. PyPI) predstavlja repozitorijum softvera i biblioteka koji se mogu koristiti pri pisanju koda. Sadrži razne biblioteke za razvoj veb-aplikacija kao što su bilioteka podrške za klijentski *HTTP* - *Requests*, *HTML* parser - *Beautiful Soup*, implementacija *SSH2* protokola - *Paramiko* i mnoge druge.

Radni okviri:
Brojni su radni okviri u ovom domenu. Mogu se podeliti na potpune radne okvire (tačnije, radne okvire za razvoj servera i klijenta) (engl. *full-stack*

framework), minijaturne radne okvire (engl. *microframework*) i asinhrone radne okvire (engl. *asynchronous framework*) [3].

Radni okviri za uporedno razvijanje servera i klijenta predstavljaju potpuno rešenje sa konfigurisanim bibliotekama koje podržavaju serverske i klijentske tehnologije. Sa druge strane, kod minijaturnih radnih okvira neretko su izostavljene podrške sistemu jezika veb šablona (engl. *web templating language*), funkcionalnosti autentifikacije i autorizacije i sl. Oni obezbeđuju neophodni minimum za pokretanje jedne veb-aplikacije, dok je ostatak prepušten programerima. Takođe, mogu da budu prevashodno orijentisani na neku konkretnu funkcionalnost (npr. asinhronost - asinhroni radni okviri).

Neki od trenutno najpopularnijih potpunih radnih okvira su *Django*, *Pyramid* i *Web2py*. Među minijaturnim radnim okvirima ističu se *Flask*, *Bottle* i *CherryPy* dok su najpopularniji *Python* asinhroni radni okviri *Tornado* i *Twisted*.



Slika 1.1: Rang lista najpopularnijih programskih, skriptovnih i jezika za označavanje

Izvor: [1]

S obzirom da se u ovom radu prevashodno bavimo asinhronim veb programiranjem, navešćemo neke glavne karakteristike asinhronih radnih okvira programskog jezika *Python*. Po pravilu oni su jednonitni (engl. *single-threaded*), što znači da ne podržavaju paralelno izvršavanje poslova. Razlog za ovakvo ograničenje je to što *Python*-ov globalni katanac za interpreter (engl. *global interpreter lock - GIL*) ne podržava paralelno izvršenje više niti. Ono što jeste podržano u ovim radnim okvirima jeste ulazno-izlazna konkurentnost (engl. *I/O concurrency*). Dakle, moguće je pokrenuti više ulazno-izlaznih operacija, konkurentno čekati njihove rezultate i obraditi ih u redosledu njihovih završetaka.

Postoje neke bitne stavke *Python*-ove standardne biblioteke koje su blokirajuće i čije bi korišćenje u (naizgled) asinhronom kodu dovele do gubitka asinhronosti. Na primer čitanje i upis u mrežne utičnice, funkcije suspenodovanja (`time.sleep()`) i sl. Zadatak asinhronog radnog okvira je da "prepravi" ove funkcije tako da one postanu neblokirajuće, ili da obezbedi zamenu za njih [6].

U ovom radu razmatraćemo asinhroni radni okvir *Tornado* programskog jezika *Python*. Zahvljujući asinhronom pristupu, radni okvir *Tornado* je izuzetno primenljiv u razvoju veb aplikacija kod kojih postoji potreba za prihvatanjem i obradom velikog broja klijenata istovremeno. Pored asinhronog pristupa, ovaj radni okvir karakteriše i ugrađeni jezik šablona koji olakšava generisanje veb sadržaja, kao i podrška za različite vidove autentifikacije i autorizacije, uz poseban akcenat na bezbednosti, o čemu će više reči biti u daljem tekstu.

Poglavlje 2

Programski jezik Python

2.1 Nastanak, istorija i filozofija

Programski jezik *Python* je delo holandskog programera Gvida van Rosuma i nastao je u laboratorijama amsterdamskog naučno-istraživačkog instituta *CWI*[3]. Decembra 1989. godine započeta je implementacija, a prvo izdanje, unutar navedenog instituta, izlazi naredne godine. Prva verzija (0.9.0) objavljena je na Juznetu (*USENET*) 20. februara 1991. godine [9].

Načelno, postoji devetnaest principa na kojima je zasnovan programski jezik.[4] Navešćemo neke od njih:

- *Eksplicitno je bolje od implicitnog.*
- *Jednostavno je bolje od kompleksnog.*
- *Kompleksno je bolje od komplikovanog.*
- *Čitljivost znači.*
- *Preko grešaka nikad ne treba preći tiho.*
- *Treba da postoji jedan – poželjno tačno jedan – očigledan način za obavljanje posla.*

Najveći uticaj na nastanak i razvoj *Python*-a imao je interpretirani programski jezik *ABC* [12], od koga je nasledio dinamičku tipizaronost i korišćenje poravnanja za razdvajanje blokova koda. Uz to, glavni tipovi podataka takođe dolaze od *ABC*-a, uz neke modifikacije.

Dakle, *Python* predstavlja interpretirani, dinamički tipizarani programski jezik, visokog nivoa, opšte namene. Podržava više programskih paradigmi poput imperativne, objektno-orijentisane i funkcionalne. Ta fleksibilnost, ali i obimna standardna biblioteka, uticali su na veliku popularnost i široku primenu ovog programskog jezika u raznim oblastima.

Vredi izdvojiti još par bitnijih datuma u dosadašnjem razvitku ovog programskog jezika. Verzija 2.0 [16] predstavljena je 16. oktobra 2000. i uz ispravku grešaka prethodnih verzija uvodi *opise lista* (engl. *list comprehension*) i

[3] *Centrum Wiskunde & Informatica*

[4] Sve one se jednim imenom nazivaju *The Zen of Python*[11]. Mogu se videti i u *Linux*-ovom *Shell*-u izvršavanjem komande `python3 -c "import this"` .

sakupljač otpadaka sa brojanjem referenci (engl. *garbage collector with reference counting*) uz algoritme detekcije ciklusa [14]. Verzija *Python 3.0* [17] (poznata i kao *Python 3000*, ili *Py3k*), nekompatabilna sa prehodnim verzijama, izlazi 3. decembra 2008. Izdanja ove verzije dolaze sa funkcionalnošću *2to3* koja delimično automatizuje proces tranzicije koda napisanog u verzijama *2.x* [15]. Počevši od 1. januara 2020. godine, grana *2.x* je i zvanično ugašena.

2.2 Tipovi podataka, funkcije i klase

Kao što je već navedeno, *Python* je dinamički tipiziran, što znači da se uslovi ograničenja tipova ne proveravaju za vreme kompilacije, već prilikom prevođenja na bajt kod. Efektivno, to znači da se tipovi ne moraju eksplicitno navoditi prilikom pisanja programa. Pored toga, *Python* je i jako tipiziran (*strongly typed*) [18]. Ovo ima za posledicu sprečavanje operacija koje nisu dobro definisane (npr. korišćenje operacije + nad brojevima i niskama u istom iskazu [5]). Podržani su sledeći ugrađeni tipovi:

tekstualni : `str`

numerički : `int, float, complex`

agregatni : `list, tuple, range`

rečnički : `dict`

skupovni : `set, frozenset`

logički : `bool`

binarni : `bytes, bytearray, memoryview`

Promenljivoj se pridružujue tip tako sto joj se dodeli vrednost tog tipa. Tip promenljive se može menjati tokom izvršavanja programa, u zavisnosti od tipa pridruzene vrednosti. Na primer, ukoliko je `x=["jabuka", "banana", "trešnja"]`, tip promenljive `x` možemo saznati funkcijom `type(x)`, što će u ovom slučaju dati rezultat `list`. Karakteristika lista, ali i torki, je da mogu sadržati vrednosti različitih tipova. Tako, prethodno navedenoj listi možemo ažurirati drugu vrednost `x[1]=2` i dobićemo potpuno legitimnu listu `x=["jabuka", 2, "trešnja"]`. Napomenimo i da su, za razliku od listi, torke *nepromenljive* (engl. *immutable*). Sličan pokušaj promene bi doveo do greške.

Bitne strukture u programskom jeziku *Python* jesu *rečnici*. Rečnici su strukture podataka indeksirani *ključem* nepromenljivog tipa. Podaci rečnika su smešteni između para vitičastih zagrada i nalaze se u formatu `ključ : vrednost`, gde su ključevi jednistveni (unutar jednog rečnika). Najvažnije operacije nad ovom strukturom podataka su čuvanje vrednosti uz dati ključ, kao i dobijanje vrednosti na osnovu ključa. Konstruktorom `dict()` pravimo rečnik od sekvence parova ključ - vrednost (Slika 2.1).

[5] Iako su izrazi `2+3` i `"hello"+"world"` dobro definisani, i daju rezultate `5` i `"helloworld"` respektivno, izraz `"hello"+1` proizvodi grešku.

```
a = dict(jedan=1, dva=2, tri=3)
b = {'jedan': 1, 'dva': 2, 'tri': 3}
c = dict(zip(['jedan', 'dva', 'tri'], [1, 2, 3]))
d = dict([('dva', 2), ('jedan', 1), ('tri', 3)])
e = dict({'tri': 3, 'jedan': 1, 'dva': 2})
```

Slika 2.1: Više mogućih načina za kreiranje rečnika koristeći konstrukt `dict()`

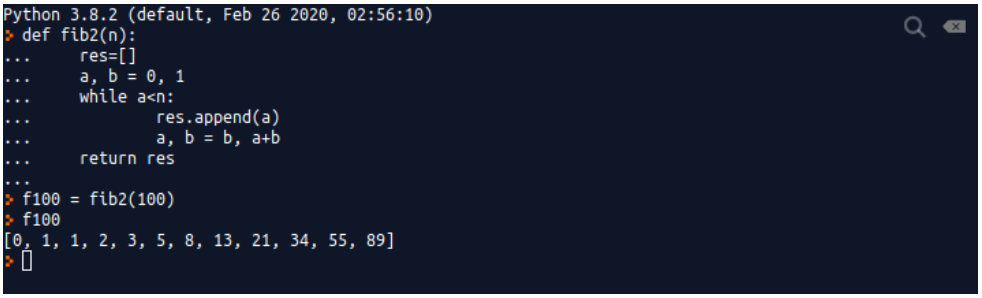
Funkcije definišemo navodeći ključnu reč `def` nakon čega sledi naziv funkcije i lista parametara unutar zagrada, praćenih dvotačkom. Telo funkcije se piše u sledećoj liniji, obavezno uvučeno čitavom dužinom (Slika 2.2).

Pri izvršavanju funkcije kreira se nova *tabela simbola* lokalnih promenljivih te funkcije. Tačnije, sva dodeljivanja vrednosti promenljivama čuvaju se unutar date tabele, te se reference ka promenljavima prvo potražuju unutar lokalnih tabela simbola, zatim unutar tabela simbola funkcija zatvorenja (*enclosing functions*), zatim u globalnim tabelama simbola i na kraju unutar tabele ugrađenih imena (*built-in names*). To ima za posledicu da se vrednosti globalnih promenljivih i promenljivih funkcija zatvorenja ne mogu menjati (osim ako se eksplicitno ne koriste ključne reči `global`, odnosno `nonlocal`), iako je referisanje na njih potpuno legitimno [19].

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> def my_function(uname):
...     print("Well, hello " + uname + ".")
...
> my_function("Mike")
Well, hello Mike.
>
```

Slika 2.2: Primer definicije, poziva i rezultata jedne jednostavne funkcije

Naredbom `return` šaljemo povratnu vrednost funkcije. Funkcija koja u sebi nema navedenu naredbu `return`, ili ima ali bez pratećeg argumenta, vraća vrednost `None`. Ova vrednost ima svoj, specijalan tip - `NoneType`. Sledeći primer (Slika 2.3) ilustruje funkciju koja računa (i vraća) listu Fibonačijevih brojeva ne većih od broja poslatog kao parametar. Uočimo iskaz `res.append(a)`. Njime se poziva metod `append()` nad listom `res`. Različiti tipovi definišu različite metode. Metodi različitih tipova mogu imati isti naziv, bez gubitka na jednoznačnosti.



Slika 2.3: Računanje vrednosti Fibonačijevog niza manjih od 100

U *Python*-u je moguće kreirati sopstvene tipove i metode koristeći klase. Klase predstavljaju način povezivanja podataka i funkcionalnosti. Kreiranjem klase, kreiramo novi tip objekta, dopuštajući kreiranje instanci tog tipa. Svaka instanca ima svoje atribute za opis stanja i metode za promenu tog stanja, koji se definišu unutar klase. Ilustrujmo navedeno jednim jednostavnim primerom (Slika 2.4). Kreiramo klasu ključnom reči `class`, nakon čega sledi naziv klase. Unutar ugrađene funkcije `__init()__` navodimo atribute klase i dodeljujemo im vrednosti. Ova funkcija se poziva svaki put po instanciranju klase. Dalje, u klasi možemo definisati sopstvene metode, ali i *nadjačati* (engl. *override*) ugrađene. Na primeru sa slike 2.4 nadjačali smo ugrađeni metod `__repr__()` koji se poziva od strane funkcije `print()`. Možemo primetiti parametar `self` u definisanim metodima. To je zapravo referenca na trenutnu instancu klase i koristi se za pristup promenljivama i metodima koji pripadaju klasi.[6] Ključnom rečju `del` možemo izbrisati atribute objektu, ali i izbrisati čitav objekat.

```
class Osoba:
  def __init__(self, ime, godine):
    self.ime = ime
    self.godine = godine

  def __repr__(self):
    return ("Zdravo! Zovem se " + self.ime + " i imam "
            + str(self.godine) + " godina.")

o1 = Osoba("Jovan", 30)
print(o1)
```

Slika 2.4: Definisanje klase Osoba

Klase programskog jezika *Python* obezbeđuju sve funkcionalnosti objektno-orijentisane paradigme. Sistem nasleđivanja klasa dozovoljava klasi da ima jednu ili više roditeljskih (baznih) klasa i izvedena klasa može da nadjača metode roditeljskih. U sledećem primeru (Slika 2.5) pravimo izvedenu klasu klase `Osoba` sa slike 2.4. Koristeći metod `super()` dobijamo objekat koji predstavlja roditeljsku klasu, tako da metode koje koristimo u izvedenoj klasi delegiramo nekoj od roditeljskih. U pratećem primeru, unutar klase `Student`, "vezali" smo

[6] Po konvenciji ovaj parametar se naziva `self`. Može se imenovati i drugačije, ali mora da stoji kao prvi parametar svake funkcije u klasi.

`__init()__` metod za prateću klasu u *redosledu rezolucije metoda* (engl *Method Resolution Order - MRO*).

```
class Student(Osoba):
  def __init__(self, ime, godine, fakultet):
    super().__init__(ime, godine)
    self.fakultet = fakultet

  def __repr__(self):
    return ("Zdravo! Zovem se " + self.ime + ", imam "
            + str(self.godine) + " godina i student sam na "
            + self.fakultet + "-u." )

s1 = Student("Jovana", 20, "MATF")
print(s1)
```

Slika 2.5: Definisanje klase Student

Svaka klasa ima svoj atribut `__mro__` koji sadrži torku referenci svih nadklasa u jedinstvenom redosledu, od trenutne klase, do klase `object` (Slika 2.6).

```
Student.__mro__
(<class '__main__.Student'>, <class '__main__.Osoba'>, <class 'object'>)
```

Slika 2.6: Ispis svih nadklasa klase `Student` – opisuje redosled nadklasa kojima se delegiraju metodi

Metod izvedene klase može da poziva metod bazne klase, čak iako imaju isti naziv [20]. Klase su u skladu sa dinamičkom strukturom *Python*-a – instance se kreiraju tokom izvršavanja programa, i mogu se menjati naknadno, odnosno, atributi se mogu dodavati objektima kasnije, izvan definicije klase. Iako legitiman, ovaj način nije preporučljiv. Bolje bi bilo u definiciji klase "nepostojećem" atributu dodeliti vrednost `None`, a kasnije mu promeniti, kada se za to ukaže potreba.

Za razliku od nekih programskih jezika objektno-orijentisane paradigme, klase u *Python*-u ne mogu da potpuno sakriju svoje atribute. Podrazumevano, oni su *javni* (engl. *public*). Ukoliko želimo da sugerišemo da je neki atribut *zaštićen* (engl. *protected*), ispred njegovog naziva treba dodati prefiks `_` (donja crta). Savesnom programeru, ovo sada daje do znanja da takvom atributu ne bi trebalo pristupati izvan klase u kojoj je definisan i njenih potklasa. Ukoliko nam je potreban *privatni* (engl. *private*) atribut, njegovom nazivu dodaćemo prefiks `__` (dve donje crte). Prevodilac, kako bi dodatno osigurao ovaj atribut, obavlja transformaciju njegovog imena u `_object._class__variable`, definitivno stavljajući do znanja programeru da takav atribut nije predviđen za pristup van klase u kojoj je definisan.

2.2.1 Funkcije kao objekti

Pre nego što nastavimo dalje, osvrnućemo se na još jednu karakteristiku funkcija programskog jezika *Python*. Kako autor navodi, na razvoj *Python*-a nisu mnogo uticali funkcionalni programski jezici. Ipak, od samog nastanka, funkcije su

objekti prve klase, što je ipak karakteristika funkcionalnih jezika [22]. Funkcije koje se tretiraju na takav način su "građani prve klase" (engl. *first class citizen*), tj. one su entiteti takvi da: [7]

- mogu biti povratne vrednosti drugih funkcija
- mogu biti prosleđene kao parametri drugim funkcijama
- se mogu dodeliti promenljivama
- se mogu čuvati u strukturama podataka poput lista, heš-mapa i sl.

Dakle, funkcije se u programskom jeziku *Python* tretiraju na isti način kao i drugi objekti. Ilustrovaćemo to jednim primerom (Slika 2.7).

```
def square(x):
  return x * x

def cube(x):
  return x * x * x

def calculate(f, x):
  return f(x)

do_square = square
do_cube = cube
```

Slika 2.7: Definicija funkcija kvadrata i kuba broja

Definišemo funkcije za računanje kvadrata i kuba broja i dodelimo ih promenljivama. Uz to, definišemo i funkciju koja primenjuje (računa) funkciju nad datom vrednošću. Sada, možemo primeniti definisane funkcije nad konkretnim vrednostima (Slika 2.8).

```
> res = calculate(do_square, 5)
> print(res)
25
> res = calculate(do_cube, 5)
> print(res)
125
>
```

Slika 2.8: Rezultati funkcija kvadrata i kuba nad brojem 5

2.2.2 Anonimne funkcije i funkcije višeg reda

Funkcija koja kao argument ima drugu funkciju, ili joj je povratna vrednost – funkcija, naziva se funkcija višeg reda (*high order function*). Koristeći ključnu reč `lambda` kreiramo anonimnu funkciju unutar jednog *Python* izraza. Anonimne funkcije predstavljaju *sintaksnu skraćenicu* (engl. *syntactic sugar*) – ispod žita kreira se funkcijski objekat isto kao kada bi se koristio `def` iskaz. Telo ovakve

[7] Termin "građani prve klase" je malo nezgrapan i može implicirati da osim njih, postoje i "manje elitne" funkcije, što ovde nije slučaj. U *Python*-u sve funkcije su prve klase.

funkcije mora biti izraz, tj. ne može da koristi druge iskaze poput `while`, `try` i sl. Najbolja primena lambda funkcija jeste upravo da posluže kao parametar neke funkcije višeg reda. Ilustrovaćemo navedeno koristeći ugrađenu funkciju višeg reda `map`[8] (Slika 2.9)

```
> a=map(lambda x: x*x, [1,2,3,4])
> list(a)
[1, 4, 9, 16]
>
```

Slika 2.9: Rezultati primene anonimne funkcije nad listom

Funkcije `map` i `filter` su i dalje prisutne kao ugrađene funkcije u *Python*-u 3, ali su nakon uvođenja *opisa lista* i *generatorskih izraza*[9] postale manje bitne. Na sledećim primerima jasno se vidi razlika u čitljivosti koda između prethodno navedenih funkcionalnosti (Slika 2.10 i Slika 2.11).

```
> list(map(square, range(6)))
[0, 1, 4, 9, 16, 25]
> [square(x) for x in range(6)]
[0, 1, 4, 9, 16, 25]
>
```

Slika 2.10: Primena funkcije kvadrata iz prethodnog primera na listu prvih pet prirodnih brojeva, uključujući i nulu

```
> list(map(square, filter(lambda x: x%2, range(6))))
[1, 9, 25]
> [square(x) for x in range(6) if x%2]
[1, 9, 25]
>
```

Slika 2.11: Primena funkcije kvadrata iz prethodnog primera, na listu svih neparnih u pet prirodnih brojeva, uključujući i nulu

Funkcija `reduce` je premeštena u modul `functools` standardne biblioteke. Iako se može koristiti, za njen uobičajeni slučaj upotrebe (sumaciju) preporučuje se upotreba funkcije `sum`. Što se tiče anonimnih funkcija, njihova primena nije posebno značajna van domena funkcija višeg reda. Usled sintaksičkih ograničenja komplikovane lambde su često nečitljive.Ipak, na dobro odabranim mestima, anonimne funkcije mogu poprilično "olakšati život" programeru i ne treba ih tek tako zanemariti.

[8] ova funkcija, uz funkcije `filter` i `reduce` može se pronaći u gotovo svim funkcionalnim jezicima (uz možda nešto izmenjena imena).

[9] Generatorski izrazi (engl. *generator expression*) predstavljaju način pisanja generatorskih funkcija – funkcija koje imaju iteratorska svojstva. [23]. Opis liste je koncizniji način za kreiranje listi, koji je više u matematičkom maniru [24].

2.3 Naredbe kontrole toka

U ovom delu poglavlja prikazaćemo naredbe za kontrolu toka programskog jezika *Python* – uslovne naredbe, petlje, funkcijske pozive, upravljanje izuzecima.

Naredba if

Programski jezik *Python* podržava višestruku `if - elif - else` naredbu za uslovno izvrsavanje blokova naredbi (Slika 2.12).

```
if x < 0:
    print("x je negativan")
elif x % 2:
    print("x je nenegativan i neparan")
else:
    print("x is nenegativan i paran")
```

Slika 2.12: Tipičan primer složene `if` naredbe

Napomenimo i da u programskom jeziku *Python*-u ne postoji `switch` naredba. Dalje, unutar samog uslova mogu stajati proizvoljni *Python* izrazi koji se mogu tumačiti u `boolean` kontekstu. Svaki ne-nula broj, neprazna niska, torka, lista ili rečnik smatra se tačnim. Isto tako, nula (bilo kog numeričkog tipa), `None` i prazna niska, torka, lista i rečnik su netačni.

Naredba while

Naredbom `while` omogućavamo ponovljeno izvršavanje naredbi ili blokova naredbi pod kontrolom uslovnog izraza. Prvo se računa vrednost uslova petlje. Ako je vrednost tačna, pristupa se izvršavanju naredbi tela petlje. Po završetku poslednje naredbe tela uslov se računa ponovo. Proces se ponavlja dokle god je uslov petlje tačan (Slika 2.13).

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Slika 2.13: Primer jednostavne `while` petlje

Postoji još načina izlaska iz petlje – naredbom `break` – kojom se izlazi iz petlje, tj. unutrašnje petlje ako imamo slučaj ugnežđenih petlji (Slika 2.14) i funkcijom `return()` – koja vraća rezultat funkcije, ukoliko se petlja nalazi unutar funkcije, terminirajući time funkciju, a samim tim i petlju.

```
while True:
  i=int(input('Unesite prirodan broj ili 0 za izlaz:\n'))
  if i==0:
    print('Izlaz iz programa')
    break
  print('Kvadrat broja', i, 'je: ', i*i, '\n')
```

Slika 2.14: Primer korišćenja naredbe `break` za izlazak iz beskonačne petlje

Naredba for

Naredbom `for` iniciramo ponavljano izvršavanje naredbe, ili bloka naredbi, pod kontrolom iterabilnog izraza[10] (Slika 2.15). Krećući se kroz kolekciju podataka, `for` naredba vezuje promenljivu za svaku vrednost iteratora redom, te se nad tom promenljivom izvršava telo naredbe (osim ako se dogodio izuzetak, `break` ili `return` naredba).

```
voce = ['mango', 'limun', 'ananas']
for v in voce:
  print('Dobar dan. Da li imate ', v, '?')
```

Slika 2.15: Primer kretanja kroz listu koristeći `for` petlju

Zanimljivo je i da će promenljiva zadržati poslednju vrednost entiteta nad kojim se iterira i nakon završetka petlje. Takođe, iteriranje kroz kolekciju koja se u samoj petlji menja, može da bude nezgodno i da da čudne rezultate. Zato je bolje prolaziti kroz kopiju same kolekcije, ili kreirati novu.

Naredbe `while` i `for` mogu, opciono, da imaju prateću `else` klauzu. Naredbe unutar `else` klauze će se izvršiti samo ukoliko se petlje završe prirodno, tj. na kraju iteriranja kroz `for` petlju i po izmeni uslova `while` petlje iz `true` u `false`, odnosno – neće se izvršiti ako su petlje prekinute "prevremeno" (koristeći `break` ili `return` naredbe i u slučaju dešavanja izuzetaka).

Obrada izuzetaka

Programski jezik *Python* podržava upravljanje izuzecima koristeći `try`, `except`, `finally` i `else` klauze. Uz to, programer može eksplicitno da podigne izuzetak naredbom `raise`. Po podizanju izuzetaka, dotadašnji kontrolni tok se zaustavlja i traži se odgovarajući *upravljač izuzecima* (engl. *exception handler*).

Po javljanju greške (tj. izuzetka), program se zaustavlja i izdaje poruku o grešci. Ovo ponašanje se može promeniti koristeći `try` - `except` komande (Slika 2.16).

[10] Više reči o iterabilnim izrazima biće u nastavku.

```
try:
  print(x)
except:
  print("Dogodlia se greska.")
```

Slika 2.16: Kako kod unutar `try` klauze generiše grešku, izvršiće se naredbe unutar `except` bloka

Moguće je definisanje višestrukih `except` blokova zarad pozivanja specifičnih blokova za specifične tipove grešaka (Slika 2.17).

```
try:
  print(x)
except NameError:
  print("Promenljiva x nije definisana.")
except:
  print("Dogodila se greska.")
```

Slika 2.17: Ukoliko promenljiva `x` nije definisana, izvršiće se prvi `except` blok; inače će se izvršiti drugi

Uz prethodno, možemo dodati i `else` blok, koji će se izvršiti samo ako nije došlo do greške u `try` bloku. Ovo može biti korisno, ukoliko imamo kod koji nije pogodan da se nalazi unutar samog `try` bloka (npr. ne želimo da ga proveravamo na date izuzetke). `finally` blok, ako je naveden, izvršiće se bez obzira na to da li se dogodila greška, ili ne. Prikladna upotreba je za zatvaranje otvorenih objekata i oslobađanje resursa.

Kao što smo već napomenuli, komandom `raise`, moguće je eksplicitno podizanje izuzetaka ukoliko su zadovoljeni odgovarajući uslovi. Dodajmo i to, da programer može definisati koji tip izuzetka želi da podigne uz odgovarajući tekst (Slika 2.18).

```
if not type(x) is int:
  raise TypeError("Dozvoljen je samo celobrojni tip.")
```

Slika 2.18: Ukoliko `x` nije celobrojnog tipa podiže se `TypeError` uz propratnu porku.

Neki od mogućih tipova grešaka su `ModuleNotFoundError`, `IndexError`, `KeyboardInterrupt`, `RuntimeError`, `ZeroDivisionError` itd.[11]

Iteratori

Prilikom definisanja `for` naredbe pominjali smo iterabilne izraze – iteratore. U narednim redovima ćemo ih opisati nešto detaljnije i uočićemo njihovu spregu sa definisanom `for` naredbom.

[11] Čitava lista, uz prateće opise, može se pronaći na [25].

Iterator je objekat programskog jezika *Python* takav da nad njime možemo pozvati metod `.next()`, koji nam daje sledeću stavku iteratora, odnosno podiže `StopIteration` izuzetak ukoliko je metod pozvan nad poslednjom stavkom. Po definisanju klasa, programer može da dopusti instancama klase da budu iterabilne tako što će definisati metod `.next()`. Uglavnom se iteratori grade implicitnim (ponekad i eksplicitnim) pozivom funkcije `iter`. Poziv generatora [23], takođe ima za rezultat iterator. Sledeći izraz (Slika 2.19) identičan je primeru sa slike (Slika 2.20) i demonstrira kako naredba `for` implicitno poziva naredbu `iter`.

```
li=[1,2,3,4,5]
for l in li:
  print(l)
```

Slika 2.19: "Klasičan" prolazak kroz listu koristeći `for` petlju

```
li=[1,2,3,4,5]
tmp_iter = iter(li)
while True:
  try:
    l = next(tmp_iter)
    print(l)
  except StopIteration:
    break
```

Slika 2.20: Prolazak kroz listu koristeći `iter` metod

Zahvaljujući iteratorima, `for` naredba se može koristiti nad kontejnerskim tipovima koji nisu nizovski (npr. rečnici), dokle god su iterabilni.

Dakle, *iterabilni* su svi objekti nad kojima možemo primeniti ugrađenu funkciju `iter()` i dobiti iterator. Oni u svojim definicijama moraju implementirati metod `__iter__`. S druge strane, *iteratori* su objekti koji imaju implementiran `__next__` metod bez argumenata koji vraća sledeći element liste i podiže `StopIteration` izuzetak, ukoliko sledeći ne postoji. Iteratori u programskom jeziku *Python* implementiraju metod `__iter__`, tako da su i oni iterabilni.

Generatori

Na više mesta do sada nailazili smo na pojam generatora i generatorskih funkcija. Generatorske funkcije predstavljaju specijalnu vrstu funkcija, koje su u stanju da vraćaju *lenje iteratore*. Preko ovakvih objekata možemo iterirati, slično kao preko listi, no za razliku od listi, lenji iteratori ne čuvaju svoj kompletan sadržaj u memoriji. Na sledećem primeru (Slika 2.21) možemo videti primer takve funkcije.

```
def lenja_lista(granica):
    index = 0
    while index < granica:
        yield index
        index += 1
```

Slika 2.21: Generisanje niza brojeva pomoću generatorske funkcije

Generatorska funkcija, prilikom poziva, vraća rezultat – generator. Ukoliko pokušamo da primenimo konstrukciju `for()` za prolazak kroz generator, dobijen od generatorske funkcije iz prethodnog primera, (što je legitimno, jer setimo se da je rezultat generatora - iterator), štampanjem rezultata dobijaćemo prirodne brojeve redom, do broja koji smo prosledili kao parametar. Ovo ne treba da nas čudi jer sada znamo da `for()` u sebi poziva `next()` kojim se računa nova vrednost generatora. Čitalac je verovatno primetio da smo u primeru uveli novu ključnu reč `yield`, koja razlikuje generatorsku funkciju od obične. Ključna reč `yield` ukazuje na mesto povratka vrednosti funkciji pozivaocu, ali za razliku od `return` naredbe koja prekida izvršenje funkcije ovde to nije slučaj. *Stanje* ove funkcije se pamti i sledećim pozivom funkcije `next()` nad generatorskim objektom nastavlja se izvršavanje ove funkcije.

Naredba pass

Telo složenog kondicionalnog izraza u *Python*-u ne može biti prazno – mora sadržati barem jedan iskaz. Naredba `pass`, koja ne izvršava nikakvu akciju, može se koristiti u tom slučaju, kada ne postoji nijedna određena akcija koju bi trebalo izvršiti (Slika 2.22)

```
if cond1(x):
    process1(x)
elif cond2(x):
    pass
elif cond3(x):
    process3(x)
else:
    proc_def(x)
```

Slika 2.22: Ilustracija `pass` komande

2.4 Python biblioteka

Python-ova standardna biblioteka je opsežni repozitorijum ugrađenih modula koji obezbeđuju pristup svim sistemskim funkcionalnostima koje bi inače bile nedostupne programerima, kao i modula koji predstavljaju standardizovano rešenje mnogih problema svakodnevnog programiranja. Uz *Python* instalaciju za *Windows* dolazi čitava standardna biblioteka uz koju su često uključene i dodatne komponente. Za operativne sisteme *Unix* tipa, *Python* sadrži kolekciju odabranih paketa, te je nekad potrebno koristiti sistemske alate zarad dobavljanja nekih (ili svih) opcionalnih komponenti. U standardnoj biblioteci se nalaze tipovi podataka koje možemo smatrati srži jezika, poput brojeva i listi. Za ove tipove, sam jezik definiše formu literala i postavlja neka ograničenja njihove semantike, ali ne definiše semantiku u potpunosti. S druge strane, jezik definiše sintaksička svojstva kao što je prioritet operatora.

Uz to, u biblioteci se nalaze i ugrađene funkcije i izuzeci – objekti koji se mogu koristiti u svim programima, bez posebne `import` naredbe.[12]

Glavni deo biblioteke sačinjava kolekcija modula. Neki od njih su napisani u *C*-u i ugrađeni su u interpreter. Drugi su pak napisani u *Python*-u i uključeni u izvornom formatu. Neki obezbeđuju interfejse i funkcionalnosti karakteristične za ovaj programski jezik, poput ispisa aktivnih *stek okvira* (engl. *stack trace*). Postoje moduli karakteristični za rad sa specifičnim operativnim sistemima, dok se razni moduli koriste za pristup i rad specifičnim aplikativnim domenima kao što je veb.

Na primer, naredbom `import datetime` u program integrišemo modul koji sadrži funkcionalnosti za upravljanje datumima i vremenom. Ukoliko bismo želeli da prikažemo trenutni datum i vreme, morali bismo da iskoristimo `datetime` objekat istoimenog modula, te bi izraz izgledao poput `print(datetime.datetime.now())`. Ova nezgrapna sintaksa može se ulepšati tako što ćemo u program direktno integrisati objekte modula koristeći konstrukt `from datetime import datetime`, te je sada za ispis trenutnog datuma i vremena dovoljno pozvati `print(datetime.now())`.

Pored standardne biblioteke, postoji rastuća kolekcija od nekoliko hiljada komponenti (od individualnih alata i modula, do čitavih radnih okruženja) dostupnih na *Python*-ovom indeksu paketa (*PyPI*). U operativnim sistemima tipa *Linux*, moguće ih je jednostavno instalirati korz komandni prozor, koristeći naredbu `pip`. Tako, radni okvir *Tornado*, dobijamo iniciranjem komande `pip install tornado`.

2.5 Radni okviri za veb

Radni okviri za razvoj veba jesu kolekcije modula i paketa koje programerima olakšavaju pravljenje veb aplikacija. Koristeći ih, programeri nemaju potrebe da eksplicitno kontrolišu svaki detalj, poput raznih komunikacijskih protokola iili upravljanja nitima i procesima, već će to okvir uraditi za njih. Rastom popularnosti programskog jezika *Python*, raste i broj radnih okvira za različite veb namene. Neki od trenutno popularnih su:

Django:
Django predstavlja potpuni radni okvir (tj. radni okvir za razvoj klijenta i servera) koji prati princip o neponavljanju (*DRY - Don't Repeat Yourself*), odnosno ideji da različiti koncepti i podaci treba da postoje na jednom i tačno jednom mestu [27]. Neke od ključnih karakteristika ovog radnog okvira su:

- Preslikavač objekata u tabele baze podataka (engl. *Object-Relational Mapper – ORM*)
- Podrška autentifikaciji
- URL rutiranje

[12] Neke od često korišćenih su `print()`, `pow()`, `super()`, `len()` itd. Sve one, uz opise i primere upotrebe, se mogu pogledati na [26].

- Generator šablona
- Shema migracija baze podataka

Flask:
Zahvaljujući svom modularnom dizajnu, ovaj radni okvir je prilično adaptivan. Pogodan za manje aplikacije, omogućava programerima da naprave dobru osnovu aplikacije, odakle se razvijanje može nastaviti u različitim pravcima. Takođe, kompatabilan je sa *Google*-ovom platformom za razvoj u oblaku (*Google App Engine*). Ključne karakteristike ovog radnog okvira su:

- Ugrađen brzi alat za pronalaženje grešaka (engl. *debugger*)
- Upravljanje *HTTP* zahtevima
- Podrška raznim ORM-ovima
- Zasnovan na junikodu (engl. *unicode*)
- Shema migracija baze podataka

Tornado:
Ovaj radni okvir, realizovan je kao još jedna asinhrona biblioteka i dizajniran je tako da se može kombinovati sa funkcionalnostima *asyncio* biblioteke [34]. Korišćenjem neblokirajućih mrežnih ulazno-izlaznih poziva ovaj radni okvir, između ostalog, omogućava i uspešno rešavanje poznatog *C10K* problema. Reč je, zapravo, o optimizaciji opsluživanja zahteva velikog broja klijenata istoveremeno. U samom nazivu krije se i broj klijenata *10K* – deset hiljada [28]. Ostale značajne karakteristike ovog radnog okvira su:

- Jezik veb šablona
- Podrška klijentskoj autentifikaciji
- Servisi realnog vremena
- Dozvoljava implementaciju autentifikacije treće strane (*3rd-party authentication*) i autorizacijske sheme

U poglavlju 4, daćemo detaljniji pregled radnog okvira *Tornado*, a u poglavlju 5 i opis veb aplikacije razvijene korišćenjem ovog radnog okvira.

Poglavlje 3

Asinhrono programiranje i podrška u jeziku Python

U uvodnom delu rada upoznali smo se sa potrebom pisanja asinhronog koda prilikom razvoja veb aplikacija, ilustrujući sve jednim kratkim primerom. Pre nego što dođemo do područja veba, u ovom poglavlju daćemo kratak pregled osnovnih koncepata asinhronog programiranja, kao i podrške koju jezik Python pruža asinhronom pristupu.

Uopšteno govoreći, za program možemo da kažemo da je asinhron ukoliko može pokretati poslove koji se mogu izvršavati van glavnog toka izvršavanja programa, pri čemu se rezultati izvršavanja tih poslova obrađuju *asinhrono* – u trenutku kada postanu dostupni. Poslovi koji se pokreću u asinhronom režimu najčešće podrazumevaju neke vremenski zahtevne operacije, poput mrežnih zahteva, ulazno izlaznih operacija i vremenskih odlaganja. Klasični blokirajući pozivi ovakvih operacija bi zaustavili glavni tok izvršavanja i procesor učinili besposlenim tokom vremena trajanja tih operacija. Prva, očigledna, poteškoća prilikom pisanja ovakvih programa je to što nam više nisu dovoljne dosadašnje sinhrone[13] funkcije, već su nam potrebne *asinhrone* funkcije – funkcije sa sposobnošću da, dok čekaju na rezultat neke asinhrone operacije, kontrolu toka prepuste drugim delovima programa koji od rezultata te asinhrone operacije ne zavise. Tokom razvoja asinhronog pristupa, ustalili su se određeni mehanizmi koje ćemo opisati u nastavku i čije je funkcionisanje omogućeno u programskom jeziku *Python*.

asinhrone funkcije – funkcije koje su u stanju da, dok cekaju na rezultat neke asinhrone operacije, kontrolu toka prepuste drugim delovima programa koji od rezultata te asinhrone operacije ne zavise."

3.1 Funkcije povratnog poziva

Funkcija povratnog poziva (engl. *callback function*) je funkcija koja se poziva automatski, kao reakcija na određeni asinhroni događaj, a koja kao argument može dobiti odgovarajuće atribute događaja koji obrađuje (npr. povratnu vrednost poziva funkcije pokrenute u asinhronom režimu). Njihovo korišćenje u domenu asinhronih aplikacija predstavlja donekle zastarelu paradigmu, ali pruža dobar uvid u to šta se zapravo dešava "iza zavese".

[13] U literaturi je čest termin i *blokirajuće*, aludirajući na to da funkcije blokiraju dalje izvršenje programa dok ne dobiju rezultat.

Pretpostavimo da u primerima koji slede radimo sa asinhronim funkcijama `sock.send()` i `sock.recv()` koje neće blokirati čitav tok izvršavanja programa. S druge strane, `on_sent` i `on_read` su funkcije povratnog poziva.

Pogledajmo primer jedne funkcije povratnog poziva (Slika 3.1). Metodu `recv()` prosledili smo dva argumenta, najveću dozvoljenu količinu podataka za prijem i funkciju koja će operisati nad rezultatom ove funkcije dohvatanja.

```
def on_read(data):
  print(data)

sock.recv(1024, on_read)
```

Slika 3.1: Dohvatanje i ispis sadržaja stranice

Na sledećoj slici (Slika 3.2) nalazi se primer ugnežđenih poziva, dakle funkcije povratnog poziva *unutar* funkcije povratnog poziva.

```
def on_sent():

  def on_read(response):
    if response.error:
      print("Error: ",response.error)

  sock.recv(1024, on_read)

sock.send(bytes('podatak'),on_sent)
```

Slika 3.2: Primer višestrukih funkcija povratnog poziva

Kako izgleda tok izvršavanja ovog koda? Na prvi pogled, očekivali bismo nešto poput `sock.send`→`on_sent`→`sock.recv`→`on_read`. No analizom ovog koda vidimo da na najvišem nivou imamo definiciju `on_sent()` funkcije i poziv funkcije `send()`. U suštini, izvršavanje kreće od poslednje linije, inicirajući neki protok podataka kroz mrežu. Kako je `sock.send()` asinhrona funkcija, ovakav program će se završiti po "prolasku" poslednje linije koda, bez obzira na to što u pozadini postoje aktivnosti koje nisu gotove. Posledica takvog pristupa bila bi da verovatno ne bismo mogli da vidimo rezultat funkcije. Ovaj problem se rešava uvođenjm entiteta koji vodi računa o ovakvim funkcijama, izvršavajući ih kada je adekvatno. Takvi entiteti se nazivaju *petlje događaja* (engl. *event loop*).

3.1.1 Petlja događaja

Petlja događaja je srž svake aplikacije koja koristi asinhroni pristup. Ona pokreće zadatke i korutine, raspoređuje funkcije ulaza i izlaza i zakazuje povratne pozive.

U zavisnosti od jezika i radnog okruženja, petlja događaja može biti pokrenuta implicitno, ili eksplicitno. Na primer, radni okvir *Node.js*, programskog jezika *JavaScript*, neće prekinuti rad programa po izvršavanju poslednje linije koda. On sadrži ugrađenu petlju događaja koja se pokreće implicitno i ulazno-izlazni metodi su podrazumevano asinhroni. Stoga će povratni pozivi

biti registrovani bez eksplicitnog pokretanja petlje. S druge strane, petlja događaja u programskom jeziku *Python* učitava se kao deo `asyncio` biblioteke i mora se pokrenuti eksplicitno.

Komunikacija sa operativnim sistemom

Generalno, posmatrano na nivou implementacije biblioteke, programski jezik je dužan da u komunikaciji sa operativnim sistemom detektuje obaveštenja o događajima (engl. *event notification*). Prilikom same implementacije biblioteke, u ovu svrhu se obično koristi odgovarajuća funkcionalnost operativnog sistema, poput *epoll* [7] u slučaju *Linux*-a, ili *kqueue* [7] u slučaju *FreeBSD*-a. Poenta je da programer, pri pisanju asinhronog koda, ne treba da brine o komunikaciji, već je dovoljno da bude svestan da po završetku asinhronih operacija dobija povratni poziv sa rezultatom koji se izvršava. Deo koji ovo sve povezuje je petlja događaja. Unutar ove petlje, koristeći odgovarajući mehanizam operativnog sistema, postavljaju se upiti operativnom sistemu o gotovim događajima, a potom se izvršavaju povratni pozivi koji su na njih čekali. Ovaj ciklus se ponavlja tokom čitavog trajanja rada programa.

Uloga petlje događaja je da nam omogući povezivanje događaja sa funkcijama povratnog poziva koje ih obrađuju. Dakle, ukoliko se tokom rada rada programa desi neki od događaja za koji je registrovana odgovarajuća funkcija povratnog poziva, petlja događaja će obezbediti pozivanje te funkcije. Ovaj mehanizam, iako prirodan u nekim jezicima poput *JavaScript*-a, u programskom jeziku *Python* dolazi tek u verziji *3.4* sa bibliotekom `asyncio` [38]. Na sledećoj slici (Slika 3.3) vidimo primer zakazivanja izvršavanja funkcije povratnog poziva.

```
def foo():
    def on_sent():
        def on_read(response):
            if response.error:
                print("Error: ", response.error)
            print(response)

        sock.recv(1024, on_read)

    sock.send(bytes('podatak'), on_sent)

if __name__ == '__main__':
    petlja = EventLoop()
    petlja.run(foo)
```

Slika 3.3: Primer izvršavanja funkcija povratnog poziva koristeći petlju događaja

Najpre smo eksplicitno napravili jednu petlju događaja i pokrenuli funkciju `foo()` u kojoj se nalazi kod sa slike 3.3. Dalje, metod `send()` inicira neki protok podataka kroz mrežu i zakazuje izvršavanje funkcije povratnog poziva `on_sent()` koja će biti pozvana kada je slanje obavljeno. Ova funkcija će pozvati funkciju `recv()` koja će registrovati funkciju povratnog poziva `on_recv()`. Ova funkcija će se izvršiti nakon uspešnog prijema podataka.

Mane funkcija povratnog poziva

Korišćenje funkcija povratnog poziva, iako je konceptualno jednostavno, u praksi ima nedostataka. Prvo, primetimo da u slučaju dešavanja greške nakon pokretanja funkcije `recv()` funkcija povratnog poziva ne bi znala da upravlja takvim izuzetkom, te se greške moraju proslediti kao objekti, a ne podizati kao izuzeci. Ovo dalje znači da, ukoliko eksplicitno ne proverimo mogućnost greške u kodu, preko njih će se preći *tiho*, što se prilično kosi sa *Python*-ovom filozofijom. Drugi problem ogleda se u tome da je jedini način da ne blokiramo funkciju povratnog poziva jeste – novim povratnim pozivom. Ovo može dovesti do velikog broja ulančanih funkcija sa povratnim pozivom nakon povratnog poziva, čime se brzo gubi na urednosti i jednostavnosti koda. U nastavku, daćemo opis još jednog načina za obezbeđivanje asinhronosti programa, bitno različit funkcijama povratnog poziva.

3.2 Korutine

3.2.1 Sintaksa yield

Prisetimo se za trenutak generatorskih funkcija iz prethodnog poglavlja. Rekli smo da one mogu da naredbom `yield` pauziraju svoje izvršavanje, i po potrebi nastavljaju. Ono što je uvedeno u *2.5* verziji *Python*-a predstavlja osnovu izgradnje korutina. Pre svega misli se na mogućnost *slanja* podataka generatoru koristeći metod `send()`. Uz ovu, dodati su i metodi `throw()` i `close()` koji dozvoljavaju pozivaocu da podigne izuzetak unutar generatora kao i da ga zatvori.

Sintaksički gledano (ne računajući nove metode), za sada se korutina ne razlikuje od generatora – to je funkcija koja sadrži ključnu reč `yield` u sebi. No, prava razlika je što sada `yield` može da se koristi sa *desne* strane znaka jednakosti. Dakle, dozvoljeno je nešto poput `podatak = yield x` gde se vrednost sa leve strane znaka jednakosti dobija slanjem vrednosti korutini, a krajnja desna vrednost je produkt korutine i vraća se pozivaocu. Ukoliko se linija završava izrazom `yield`, vraća se – `None`. Pogledajmo sledeći primer (Slika 3.4):

```
def korutina(a):
    print('-> Started: a =', a)
    b = yield a
    print('-> Received: b =', b)
    c = yield a + b
    print('-> Received: c =', c)

k = korutina(14)
print(next(k))
print(k.send(28))
print(k.send(99))
```

Slika 3.4: Primer kreiranja i pokretanja jedne jednostavne korutine

Kreiramo korutinu `k` i šaljemo joj vrednost `14`. Pozivom `next(k)` korutina se izvršava do prvog `yield`-a, ispisujući tekst na standardni izlaz i vraća nam

vrednost `a` koju potom ispisujemo. Pozivom `k.send(28)`, ponovo pokrećemo izvršavanje korutine, šaljući joj vrednost, koju ona sada dodeljuje promenljivoj `b` i nastavlja sa svojim radom. Analogno ovome, izvršava se ostatak korutine. Na sledećoj slici (Slika 3.5) možemo videti rezultat ovih poziva.

Slika 3.5: Rezultati poziva prethodno definisane korutine

Vredi napomenuti i sledeće. Kako će argument metoda `send()` postati vrednost pauzirajućeg `yield` zahteva, to znači da se ovakvi metodi ne mogu pozivati pre nego što se korutina suspenduje, tj. ne mogu biti pozvani pre poziva `next()` ili se korutini ne pošalje "prazna vrednost" (`k.send(None)`). Ovaj sistem se naziva *priprema korutine* (engl. *priming the coroutine*). Postoji način koji nam omogućava da ovo zaobiđemo, a to je upotreba dekoratora. Stavljanjem dekoratora `@coroutine` pre same definicije korutine, dobijamo pripremljenu korutinu kojoj odmah možemo slati vrednosti.[14]. Sintaksa `yield from`, koju ćemo prikazati u nastavku, automatski vrši i prpremu korutine, te je nekompatabilna sa dekoratorom `@coroutine`.

3.2.2 Povratne vrednosti

Pre verzije *Python 3.3*, korutine nisu mogle imati povratne vrednosti (mislimo na naredbu `return`). Međutim, ovo se pokazalo kao izuzetno korisna karakteristika. Pogledajmo sledeći primer (3.6):

```
def prosek():
    ukupno = 0.0
    broj = 0
    prosecna_vrednost = None
    while True:
        vr = yield
        if vr is None:
            break
        ukupno += vr
        broj += 1
        prosecna_vrednost = ukupno/broj
    return (broj, prosecna_vrednost)
```

Slika 3.6: Korutina za računanje proseka

Nakon kreiranja i pripreme, korutini šaljemo različite celobrojne vrednosti. Slanjem vrednosti `None`, petlja se završava vraćajući rezultat – prosečnu vrednost svih poslatih brojeva. Kao i obično, kraj generatora podiže `StopIteration`

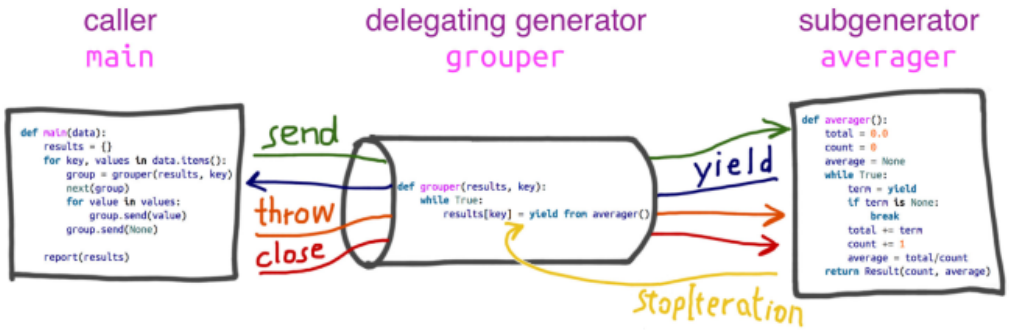
[14]Brojni radni okviri obzbeđuju specijalne dekortaore za rad sa korutinama. Uloga takvih dekoratora je raznovrsna i ne služe svi za pripremu korutine. Na primer, u radnom okviru *Tornado*, dekorator `@gen.coroutine` povezuje korutinu sa petljom događaja.

izuzetak [41]. Ukoliko bismo hteli da se naš program nastavi po ovakvom prekidu korutine mogli bismo da obradimo ovaj tip izuetka i u `except` klauzi sačuvamo vrednost. Drugi i smisleniji način je (ponovo) obezbeđen konstruktom `yield from`. Osim što interno obrađuje ovakav izuzetak, vrednost atributa ovog izuzetka postaje vrednost samog `yield from` izraza.

3.2.3 Sintaksa yield from

Prva stvar koju valja odmah reći je da je `yield from` potpuno novi konstrukt. Njime je omogućena primena `yield` izraza nad drugim generatorom (odnosno korutinom). Kada generator `gen()` uradi nešto poput `yield from subgen()`, novi podgnerator `subgen()` preuzima kontrolu i svoje vrednosti (komandom `yield`) šalje *pozivaocu generatora* `gen()`. Slično, pozivaoc će komunicrati sa podgeneratorom `subgen()` direktno. Početni generator `gen()` će biti blokiran do završetka rada podgeneratora.

Glavna uloga komande `yield from` je da generiše dvosmerni kanal od spoljašnjeg pozivaoca, do unutrašnjeg podgeneratora, tako da se vrednosti mogu kretati između njih direktno, a izuzeci podizani bez dodavanja njihove obrade u središnjim korutinama. Efektivno, ovo omogućava delegaciju korutina, kakva nije bila moguća ranije. Pogledajmo sledeći primer (Slika 3.7).



Slika 3.7: Sistem propagacije korutina za računanje proseka

Izvor: [39]

Dakle, središnji generator (Slika 3.8) je suspendovan komandom `yield from`, te pozivalac (Slika 3.9) i unutrašnji generator (korutina za računanje proseka sa slike 3.6) direktno komuniciraju komandama `send()` i `yield`. Kontrola se vraća središnjem generatoru kada se podgenerator vrati i interpreter podigne `StopIteration` izuzetak.

```
def grouper(results, key):
  while True:
    results[key] = yield from averager()
```

Slika 3.8: Delegirajući generator

```
def main(data):

  results = {}
  for key, values in data.items():
    group = grouper(results, key)
    next(group)
    for value in values:
      group.send(value)
    group.send(None)

  print(results)
```

Slika 3.9: Pozivalac generatora

3.3 Biblioteka asyncio

Na početku ovog poglavlja pominjali smo ideju o funkcijama koje pauziraju svoje izvršavanje i nastavljaju u pogodnom momentu, uz mogućnost međusobne komunikacije, i načina za lepu obradu izuzetaka. Upravo smo tako nešto i dobili. U verziji *Python 3.4* izlazi `asyncio` biblioteka i naše korutine dobijaju novo značenje. Ovde pre svega mislimo na to da korutina u okviru ove biblioteke *mora* da koristi `yield from` (za razliku od malopređašnjih, gde je i `yield` bilo legitimno). Drugo, jedna korutina može biti pozvana od strane druge (koristeći, ponovo, `yield from`, ili koristeći neku funkciju ove biblioteke poput `asyncio.run()`). Na posletku, navedimo i da bi dekorator `@asyncio.coroutine` trebalo da stoji pre same definicije korutine.

Biblioteka `asyncio` olakšava proces pisanja asinhronog koda koristeći korutine. Koristi se kao osnova za više asinhronih radnih okvira koji se koriste u domenu komunikacija – za implementaciju veb-servera, konekcija ka bazama podataka, mreža velikog opterećenja i dr. Ona predstavlja dobar izbor kod mrežnih zadataka sa čestom korisničkom interakcijom.

U okviru biblioteke dostupni su interfejsi viskog nivoa za konkurentno pokretanje korutina i kontrolu njihovog izvršavanja, kao i dodatni mehanizmi sinhronizacije, raspoređivanja zadataka, kontrolu podprocesa i sl. Uz to, dostupni su i interfejsi niskog nivoa za kreiranje i kontrolu petlji događaja, povezivanja delova programa koji koriste funkcije povratnog poziva, sa delovima programa koji koriste korutine, ali i mnogi drugi.

Kao što vidimo, ovo je jedna moćna biblioteka za razvoj asnihronih aplikacija u programskom jeziku *Python*. Pogledajmo jedan primer (Slika 3.10) programa koji je zasnovan na korutinama.

```python
@asyncio.coroutine
def spin(msg):
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            yield from asyncio.sleep(.1)
        except asyncio.CancelledError:
            break
    write(' ' * len(status) + '\x08' * len(status))

@asyncio.coroutine
def slow_function():
    yield from asyncio.sleep(3)
    return 42

@asyncio.coroutine
def supervisor():
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = yield from slow_function()
    spinner.cancel()
    return result

def main():
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(supervisor())
    loop.close()
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

Slika 3.10: Primer korišćenja korutina u biblioteci `asyncio`

Izvor: [39]

Sam program ne radi nešto specijalno. Korisniku, u konzoli, prikazuje "animaciju" čekanja[15], dok se ne generiše rezultat (`42`). Na više mesta u primeru koristili smo `asyncio.sleep()`. Ukoilko bismo na tim mestima koristli `time.sleep()` potpuno bismo izubili na asinhronosti, jer sistemski poziv `time.sleep()` blokira petlju događaja. Metod `asyncio.create_task()` prihvata korutinu i od nje pravi *zadatak* (engl. *task*) zakazujući njegovo izvršavanje. Odmah po kreiranju zadatka dobijamo *nekakav* rezultat. Ispisivanjem tog rezultata (u narednoj liniji) dobijamo nešto poput `spinner object: <Task pending name='Task-2' coro=<spin() running at main.py:5>`. U `main` funkciji kreiramo petlju događaja i zakazujemo izvršavanje korutine. Povratna vrednost korutine je ujedno i povratna vrednost ovog poziva. Napomenimo i da zadatke ne instanciramo sami, već ih dobijamo šaljući korutinu nekom od metoda `asyncio` biblioteke. Ukoliko dobijemo takav objekat, to znači da je on već zakazan za izvršavanje na petlji događaja. Metodom `cancel()` zadatak možemo prekinuti, što ima za posledicu podizanja `CancelledError` izuzetka unutar korutine.

[15] Kako je i program konzolni, to je i animacija jako niskog nivoa. Brzim smenjivanjem karaktera `|`, `/`, `-`, `\` dobijamo privid okretanja crtice.

U prethodnom primeru smo ilustrovali tri različita načina izvršavanja korutina. Jedan je bio korišćenjem sistemskog poziva `run_until_complete()` koji obezbeđuje biblioteka. Drugi je bio primenom `yield from` konstrukta nad samom korutinom, dok je treći ilustrovan pravljenjem zadatka. U svakom slučaju, po detektovanju čekanja, korutina se suspenduje i pokreće se naredna.

3.4 Poređenje korutina i funkcija povrtanog poziva

Funkcije povrtanog poziva predstavljaju tradicionalan način implementacije asinhronih poziva. Kao što smo videli ranije, to je jedan jednostavan koncept kojim umesto da čekamo na odgovor, zakazujemo funkciju koja će se pozvati nakon određene operacije. Naravno, ovaj pristup je moguć samo ukoliko imamo neku petlju događaja koja će voditi računa o tome da se ono zakazano zaista i izvrši. Tako petlja, a sa njom i ostale funkcije, neće biti blokirana, pod uslovom da mi, kao programeri, ne pravimo greške.

S druge strane, koristeći korutine (odnosno generatore) dobijamo alternativni način pisanja asinhronog koda. Iz perspektive petlje događaja, pokretanje funkcije povratnog poziva ili "buđenje" korutine metodom `send()` nema preterane razlike. Pravu razliku treba tražiti iz ugla programera. Nije retkost naići na višestruke ugnežđene povratne pozive, gde jedna operacija zavisi od rezultata prethodne (Slika 3.11).

```
def faza1(odgovor1):
  zahtev2 = korak1(odgovor1)
  poziv_apiju_2(zahtev2, faza2)

def faza2(odgovor2):
  zahtev3 = korak2(odgovor2)
  poziv_apiju_3(zahtev3, faza3)

def faza3(odgovor3):
  korak3(odgovor3)

poziv_apiju_1(zahtev1, faza1)
```

Slika 3.11: Primer nadovezanih funkcija povratnog poziva

Kod slike 3.11 je prilično težak za čitanje i još teži za pisanje, dok verovatno najveća komplikacija leži ukoliko se mora otkloniti neka kasno uočena greška. Svaka od ovih funkcija uradi deo posla, postavi funkciju povratnog poziva i vrati kontrolu petlji događaja. U tom trenutku, sav lokalni sadržaj je izgubljen. Kada se, recimo `faza2` izvrši, izgubli smo vrednost `zahtev2`. Ako nam je `zahtev2` neki bitan podatak, moramo se oslanjati na globalne promenljive za skladištenje između poziva. U tom momentu, korutine pokazuju jasan napredak. Unutar korutine, za obavljanje tri različita asinhrona poziva dovoljno je primeniti `yield from` konstrukt na tri mesta, vraćajući kontrolu petlji događaja svaki put kada se za to ukaže potreba. Kada je rezultat spreman, korutina se aktivira metodom `send()`. Opet, posmatrajući iz perspektive programera, imamo tri operacije

unutar tela jedne funkcije, što prilično liči na klasičan, imperativni način programiranja. (Slika 3.12).

```
@asyncio.coroutine
def korutina(zahtev1):
    odgovor1 = yield from poziv_apiju_1(zahtev1)

    zahtev2 = korak1(odgovor1)
    odgovor2 = yield from poziv_apiju_2(zahtev2)

    zahtev3 = korak2(odgovor2)
    odgovor3 = yield from poziv_apiju_3(zahtev3)
    korak3(odgovor3)

loop = asyncio.get_event_loop()
loop.create_task(korutina(zahtev))
```

Slika 3.12: Primer uzastopnog poziva korutina

Pretpostavimo da je došlo do podizanja *I/O* izuzetka u procesiranju poziva `poziv_apiju_2(zahtev2, faza2)` sa slike 3.11. Izuzetak ne može biti uhvaćen u funkciji `faza1()` jer je `poziv_apiju_2` asinhroni poziv. On se vraća odmah, pre nego što su primenjene ikakve operacije ulaza ili izlaza. U svetu asinhronih eksternih interfejsa (kao što bi bio ovaj), ovo je rešeno registrovanjem dve funkcije povratnog poziva za svaki asinhroni poziv. Jedna obrađuje rezulatat u slučaju uspeha, dok je druga za obradu grešaka. S druge strane, ukoliko na primeru sa slike 3.12, dođe do podizanja izuzetka u nekom od asinhronih poziva, takav izuzetak će biti uhvaćen smeštanjem `yield from` konstrukata u `try/except` blokove.

No, nije ni ovakav pristup bez mana. Pre svega, treba dobro shvatiti koncept korutina i navići se na funkcionisanje konstrukta `yield from`. Dalje, treba biti svestan da korutine ne možemo pozivati kao obične funkcije (kao što smo mogli `poziv_apiju_1(zahtev1, faza1)` da bismo inicirali seriju povratnih poziva). Mora se eksplicitno zakazati izvršavanje korutine na petlji događaja, ili "aktivirati" korutinu korišćenjem `yield from` naredbe unutar druge korutine koja je zakazana za izvršavanje.

3.5 Sintaksa async/await

Sledeća verzija *Python 3.5* sa sobom donosi neke novitete `asyncio` biblioteke, uvodeći novu sintaksu i uz nju objekte koji su u stanju da *asinhrono čekaju* (engl. *awaitable*), kao i mogućnosti *asinhrone iteracije* i *asinhrone upravljače kontekstom* [40].

Što se tiče nove sintakse, pre svega, misli se na definisanje korutina korišćenjem ključnih reči `async` i `await`.[16] Tokom daljeg razvoja programskog jezika *Python*, ovakav način definisanja korutina (pa i pisanja asinhronih aplikacija) pokazao se kao najpoželjniji. Jedan kratak primer nalazi se na slici (Slika 3.13).

[16] Tehnički gledano, u tom trenutku `async` i `await` još nisu bile *prave* (rezervisane) ključne reči, već se koriste za označavanje i suspendovanje korutina. U ovoj verziji još smo mogli da koristimo ove reči za imenovanje funkcija i promenljivih. Rezervišu se u verziji *Python 3.7*.

Čitaocu je verovatno jasno, ali nije zgoreg ponoviti, da iako koristimo nešto drugačiji konstrukt, "klasično" pozivanje korutine (na način na koji bismo pozvali običnu funkciju) ne podrazumeva njeno izvršenje.

```
import asyncio

async def func():
  print("Zdravo...")
  await asyncio.sleep(1)
  print("...svete")

asyncio.run(func())
```

Slika 3.13: Program ispisuje "Zdravo...", pauzira jedan sekund i ispisuje "...svete"

Primetimo `await` konstrukt unutar korutine. Njime suspendujemo izvršavanje korutine dok rezultat ne bude dostupan. Pomenuli smo i objekte koji mogu da asinhrono čekaju. To su korutine, ali i bilo koji objekti koji u sebi imaju definisan `__await()__` metod. U prethodnom primeru koristili smo i sistemski poziv `run` biblioteke `asyncio`. Ovo je poziv visokog nivoa koji izvršava korutinu, kreirajući i vodeći računa o petlji događaja.

Novom sintaksom nismo izgubili stare načine pokretanja korutina. I dalje možemo praviti zadatke (Slika 3.14), ali i primenjivati `await` konstrukt nad samom korutinom (Slika 3.15).

```
import asyncio

async def pisi_nakon(pauza, izraz):
    await asyncio.sleep(pauza)
    print(izraz)

async def main():
    zad1 = asyncio.create_task(
        pisi_nakon(1, 'Zdravo...'))

    zad2 = asyncio.create_task(
        pisi_nakon(2, '...svete'))

    await zad1
    await zad2

asyncio.run(main())
```

Slika 3.14: Program pokreće prvi zadatak, detektuje čekanje, pokreće drugi zadatak i ispisuje dati tekst prvog, pa drugog zadatka

Ovaj način omogućava konkurentno izvršavanje korutina. Dakle, ukoliko u zadatku nailazimo na čekanje, zadatak suspendujemo i pokrećemo naredni zadatak. Po završavanju čekanja nastavljamo dalje sa zadacima. U primeru sa slike 3.14 ukupno čekanje bi bilo dve sekunde.[17]

[17] Napomenimo i da, počevši od verzije 3.8 *Python* uvodi metod `gather()` ove biblioteke. Njemu možemo, kao parametre, proslediti korutine i dobićemo jedan `Future` objekat. Izvršavanjem ovog objekta pokrećemo sve korutine konkurentno i dobijamo rezultat identičan pravljenju i pokretanju zadataka.

```
import asyncio

async def fun2():
    return 7

async def fun1():
    print(await fun2())

asyncio.run(fun1())
```

Slika 3.15: U korutini `fun1` pozivamo korutinu `fun2` i ispisujemo rezultat — 7

Postoji još jedan tip objekata koji je u stanju da asinhrono čeka. To su instance klase `asyncio.Future`. Uz ovaj, u programskom jeziku *Python*, postoji i sličan objekat u biblioteci `concurrent.futures.Future`[18]. Iako imaju slične interfejse, postoji razlika u njihovim implementacijama. Ovo su objekti niskog nivoa koji predstavljaju eventualni rezultat neke asinhrone operacije. Kada primenimo konstrukt `await` nad ovim objektom, korutina će pauzirati svoje izvršavanje dok se objekat ne realizuje. Slično svom pandanu `yield from` iz ranije verzije, i `await` konstrukt se ne može primeniti van korutine. Upotrebom ovih objekata, možemo da dozvolimo korišćenje konstrukta `await` programima koji koriste funkcije povratnog poziva. Načelno, ne postoji potreba za njihovim eksplicitnim kreiranjem, već se treba oslanjati na poziv `loop.create_future()` kojim dobijamo objekat zakačen za petlju događaja (Slika 3.16).

```
async def set_after(fut, delay, value):
    await asyncio.sleep(delay)
    fut.set_result(value)

async def main():
    loop = asyncio.get_running_loop()
    fut = loop.create_future()
    loop.create_task(
        set_after(fut, 1, '... world'))
    print('hello ...')
    print(await fut)

asyncio.run(main())
```

Slika 3.16: Primer kreiranja *Future* objekta i postavljanja njegove vrednosti

Treba napomenuti da mnoge funkcije same vraćaju *Future* objekte. Kao što ćemo videti u nastavku, u radnom okviru *Tornado*, *Future* objekti predstavljaju povratne vrednosti jako velikog broja funkcija.[19]

[18] Povodom ove nezgode u označavanju, u *PEP*-u 3156 [38] rečeno je da postoji mogućnost da u bliskoj budućnosti vidimo spajanje ovih klasa implementiranjem metoda `__iter__`, koji će raditi sa `yield from`, u "konkurentnoj" klasi.

[19] Ovo se odnosi na verzije nakon *Tornado 5.0*. Pre toga, funkcije su mahom koristile povratne pozive za obezbeđivanje asinhronosti.

U narednim izdanjima programskog jezika *Python* sintaksa `async/await` postaje dominantna u pisanju korutina.[20] Zato, korutine pisane ovim stilom nazivamo *prirodne korutine* (engl. *native coroutine*), razlikujući ih od prethodnih – dekorisanih korutina. Koristeći izraze `async for` (asinhrona iteracija) i `async with` (asinhroni upravljači kontekstom) neki obrasci postaju jednostavniji za implementaciju.

3.5.1 Async/await u drugim programskim jezicima

Pre nego što pređemo na detaljniji pregled radnog okvira *Tornado*, napomenućemo da se asinhroni pristup pojavljuje i u drugim programskim jezicima. Specijalno, sintaksa `async/await`, koja značajno pojednostavljuje pisanje asinhronih programa, ima svoju implementaciju u jezicima *C#*, *JavaScript*, *Kotlin*, *Hack*, *Dart*, a uz pomoć nekih ekspoerimentalnih biblioteka i ekstenzija i u programskim jezicima *Scala* i *C++*. Tako, na sledećem primeru (Slika 3.17) vidimo primer jedne asinhrone funkcije programskog jezika *C#*.

```
public async Task<int> FindPageSize(Uri uri) {
  byte[] data = await new WebClient()
  .DownloadDataTaskAsync(uri);
  return data.Length;
}
```

Slika 3.17: Primer kreiranja *Future* objekta i postavljanja njegove vrednosti

Prvo, ključna reč `async` označava ovu funkciju kao asinhronu, te se u njoj može pojavljivati jedan (ili više) `await` izraza. Povratni tip `Task<T>` je sličan konceptu *Future*-a, s tim što se ovde i ukazuje da će stvarni povratni tip biti celobrojna vrednost. Prvi metod koji se izvršava u ovoj funkciji je `new WebClient().DownloadDataTaskAsync(uri)`, što takođe predstavlja asinhroni metod koji, kao povratnu vrednost ima `Task<byte[]>`. Kako je metod asinhron, to neće preuzeti čitav dokument pre vraćanja, već će započeti proces preuzimanja, i odmah vratiti `Task<byte[]>` objekat. Kada je preuzimanje završeno, zadatak se *razrešava* (engl. *resolve*) sa primljenim podacima. Ovo dalje inicira povratni poziv i omogućava početnoj funkciji `FindPageSize()` da nastavi sa radom, vezujući datu vrednost za promenljivu `data`.

[20] Trenutna verzija, u vremenu pisanja ovog rada, je verzija *Python 3.8*. Generisane korutine se tretiraju kao zastarele i zakazano je njihovo izbacivanje u verziji *Python 3.10*.

Poglavlje 4

Radni okvir Tornado

4.1 Nastanak i primena

Tornado je radni okvir programskog jezika *Python* i biblioteka za asinhroni razvoj aplikacija, prvobitno razvijan za *FriendFeed*[21] [29]. Koristeći funkcionalnosti neblokirajućeg ulaza i izlaza, aplikacije razvijane kroz *Tornado* mogu da podrže veliki broj istovremeno otvorenih konekcija, čineći ga idealnim a razvoj aplikacija koje koriste *dugotrajno anketiranje* (engl. *long polling*), *veb utičnice* (engl. *web sockets*) i ostalih vidova dugotrajnih konekcija sa korisnicima.

Ovaj radni okvir se načelno sastoji od četiri različite komponente:

- Radni okvir za razvoj veb aplikacija (izvođenjem potklasa klase `RequestHandler` i dodavanjem drugih klasa podrške)
- Klijentska i serverska podrška *HTTP* zahtevima kroz `AsyncHttpClient` i `HTTPServer`
- Asinhrona biblioteka sa klasama `IOLoop` i `IOStream`
- Biblioteka za rad sa prirodnim korutinama

Koristeći prve dve komponente, *Tornado* pruža programerima alternativu *interfejsu prolaza veb servera* (engl. *Web Server Gateway Interface - WSGI*), dok su druge dve zadužene za uspostavljanje asinhronosti servera.

Kako autori navode, *Tornado* je pravljen nad bibliotekom `asyncio` sa kojom deli petlju događaja. Funkcionalnosti obezbeđene u ovom radnom okviru mogu se kombinovati sa funkcionalnostima `asyncio` biblioteke.

4.2 Asinhroni i blokirajući pozivi

U delokrugu veb aplikacija u realnom vremenu, kao i drugih aplikacija koje zahtevaju *dugotrajne* (engl. *long-lived*) konekcije, potrebno je uspostaviti konekciju sa svakim pojedinačnim korisnikom, koja može značajan deo svog vremena da bude *nekorišćena* (engl. *idle*). U tradicionalnim (sinhronim) veb serverima to

[21] *FriendFeed* je bila socijalna mreža, čiji su korisnici mogli da dobijaju sadržaj u realnom vremenu. Sam sadržaj je bio sačinjen od više celina, poput korisničkih objava, vesti, blogova, platformi za fotografije i sl. *FriendFeed* je kupljen od strane *Facebook*-a 10. avgusta 2009. godine.

znači da se svakom korisniku posvećuje jedna nit, što može biti jako skupo u slučaju velikog broja istovremeno uspostavljenih konekcija.

Koristeći jednonitnu petlju događaja, *Tornado* značajno smanjuje cenu konkurentnih konekcija. Kako u svakom trenutku možemo izvršavati najviše jednu operaciju, programeri treba da teže ka pisanju asinhronog koda.

Asinhrone operacije u *Tornado*-u interno vraćaju *Future* objekte, sa izuzetkom nekih komponenti poput `IOLoop` koja koristi funkcije povratnog poziva. *Future* objekti se transformišu u svoje rezultate korišćenjem ključnih reči `await` ili `yield`.

U sledećem primeru (Slika 4.1) nalazi se sinhrona funkcija koja dohvata sadržaj strane sa datog *url*-a i njen asinhroni pandan (Slika 4.2).

```
from tornado.httpclient import HTTPClient

def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

Slika 4.1: Sinhroni način dovlačenja sadržaja koristeći *Tornado*-v `HTTPClient`

```
from tornado.httpclient import AsyncHTTPClient
import asyncio

async def asynchronous_fetch(url):
    http_client = AsyncHTTPClient()
    response = await http_client.fetch(url)
    return response.body
```

Slika 4.2: Asinhroni način dovlačenja sadržaja koristeći *Tornado*-v `AsyncHTTPClient`

Kao što smo i naveli ranije, asinhrone manipulacije interno vraćaju *Future* objekte, te se primer sa slike 4.2, zapravo svodi na nešto poput sledećeg primera (Slika 4.3).

```
from tornado.httpclient import AsyncHTTPClient
from tornado.concurrent import Future
import asyncio

def async_fetch_manual(url):
    http_client = AsyncHTTPClient()
    my_future = Future()
    fetch_future = http_client.fetch(url)
    def on_fetch(f):
        my_future.set_result(f.result().body)
    fetch_future.add_done_callback(on_fetch)
    return my_future
```

Slika 4.3: Interpretacija `async`/`await` načina, klase `AsyncHTTPClient()`

U radnom okviru *Tornado*, najpoželjniji način pisanja asinhronog koda jesu upravo korutine. One koriste *Python*-ove ključne reči `await`, odnosno `yield`, za suspenodavanje i ponovno nastavljanje izvršavanja.

4.3 Struktura Tornado veb aplikacije

Tornado veb aplikacija se, u opštem slučaju, sastoji se iz jedne ili više potklasa klase `RequestHandler` u kojima se definišu metodi za generisanje i obradu sadržaja veb stranica, aplikacijskog objekta (instanca klase `tornado.web.Application`) koji prosleđuje dolazeće zahteve odgovarajućim *rukovaocima* (engl. *handlers*) i `main()` funkcije koja startuje server (Slika 4.4).

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

Slika 4.4: Primer minmimalne Tornado veb aplikacije

Nakon uključivanja neophodnih biblioteka, izvodimo jednu rukovalačku potklasu i u njoj definišemo `get` metod koji se poziva po detekciji *GET* zahteva klijentovog pregledača. Metodom `write` generišemo sadržaj tražene veb stranice. U nešto komplikovanijim primerima, ovaj metod biće zamenjen metodima koji rade sa *šablonima* (engl. *templates*). Dalje, definišemo funkciju koja generiše instancu aplikacijskog objekta, sa navedenim podešavanjima. U ovom slučaju, naš rukovalac je zadužen za indeksnu stranicu. Nakon toga, unutar funkcije `main()`, pokrećemo server dodeljujući instanci aplikacije datu adresu i port broj (ukoliko je adresa izostavljena, uzima se `"localhost"` tj. `"127.0.0.1"`). Za uspešno pokretanje servera, neophodna je poslednja linija. `IOLoop` je omotač (engl. *wrapper*) oko petlje događaja biblioteke `asyncio`. Metodom `current()` vraćamo trenutnu petlju koju pokrećemo metodom `start()`.

Aplikacijski objekat

Aplikacijski objekat zadužen je za globalnu konfiguraciju, uključujući i tabelu rutiranja koja pridružuje veb-adresama stranica rukovaoce koji su za njih zaduženi. Dakle, tabela sadrži listu torki koje su sačinjene od regularnog izraza koji opisuje strukturu putanje i naziva rukovaoca koji je toj putanji pridružen. Redosled

navođenja je bitan, jer se prilikom pretrage za rukovaocem poziva rukovalac prvog u nizu prepoznatog izraza. Od ostalih konfiguracionih elemenata, čiji broj nije mali, navešćemo samo neke, poput `template_path` koji označava putanju ka direktorijmu sa šablonima, `static_path` – putanja ka direktorijmu sa statičkim fajlovima, `cookie_secret` – vrednost koja se postavlja za označavanje kolačića (engl. *cookie*) itd.

Nasleđivanje klase RequestHandler

Najveći deo posla jedne veb aplikacije, *Tornado* obavlja kroz potklase klase `RequestHandler`, odnosno u njihovim metodama imenovanim po *HTTP* metodama koje obavljaju – `get()`, `post()` i sl. Svaki rukovalac definiše metode koji su mu potrebni, odnosno, koji su potrebni stranici za koju je on zadužen. U okviru tih metoda izvršava se odgovarajuća serverska logika, a zatim se, koristeći pozive poput `RequestHandler.render` ili `RequestHandler.write`, generišu stranice koje se šalju klijentu kao odgovor na zahtev. Praksa je i da se napravi jedna, glavna, klasa *BaseHandler* (inače takođe potklasa `RequestHandler`-a) u kojoj će biti nadjačani metodi poput `write_error`, ili `get_current_user`, a koju će potom naslediti ostale rukovalačke klase. Uz to, rukovalac može da pristupi objektu poslatog zahteva pomoću `self.request`, a podaci iz *HTML* formulara su parsirani i moguće ih je dohvatati metodama poput `get_query_argument` i `get_body_argument`. Prilikom svakog zahteva upućenog serveru, izvršava se naredna sekvenca poziva:

1. Kreira se novi `RequestHandler` objekat
2. Poziva se metod `initialize()` sa argumentima iz konfiguracije `Application` objekta. Kako ovaj metod ne generiše izlaz, to se na ovom mestu uglavnom poslati argumenti smeštaju u promenljive.
3. Poziva se metod `prepare()`. Najbolje je definisati ovaj metod u baznoj klasi (*BaseHandler*) pošto će se on izvršiti bez obzira na to koji je *HTTP* metod iniciran. Uz to, on može i da genereiše izlaz – ukoliko poziva `finish` (ili `redirect`) dalje procesiranje se zaustavlja.
4. Poziva odgovarajući HTTP metod: `get()`, `post()`, `put()` i sl.
5. Po završetku zahteva poziva se metod `on_finish`, koji obavlja završne akcije tipa čišćenja, odjavljivanja i sl.

Određene metode rukovaoca, kao što su `prepare()` i *HTTP* metodi `get()`, `post()` i sl. mogu se pretvoriti u korutine, kako bi rukovalac bio asinhron (Slika 4.5).

```
class MainHandler(tornado.web.RequestHandler):
  async def get(self):
    http = tornado.httpclient.AsyncHTTPClient()
    response = await http.fetch("https://www.google.com/")
    print(response)
```

Slika 4.5: Asinhroni `get()` metod

4.4 Formulari i šabloni

U prethodnoj sekciji upoznali smo se sa osnovnom strukturom jedne veb aplikacije koristeći funkcionalnosti radnog okvira *Torado*. U nastavku, navešćemo neke moćne mehanizme koji olakšavaju rad pri konstruisanju iole ozbiljnijih veb aplikacija. Kao i drugi radni okviri za veb, i *Torado* ima za cilj da programerima omogući brže pisanje aplikacija, omogućavajući im da ponovo iskoriste što više prethodno napisanog koda. Jedan od mehanizama koji ovo omogućava je ugrađeni jezik veb šablona kroz `tornado.template` modul.

Podrazumevano, *Tornado* će tražiti fajlove šablona u istom direktorijumu gde se nalaze i `.py` fajlovi koji referišu na njih. Ipak, urednije je grupisati ih sve u jedan direktorijum i iskoristiti `template_path` za njihovo referisanje.

Sintaksa šablona

Šablon predstavlja *HTML* kod sa umetnutim *Python* izrazima i kontrolnim sekvencama na odgovarajućim mestima (Slika 4.6).

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <ul>
      {% for item in items %}
        <li>{{ item }}</li>
      {% end %}
    </ul>
  </body>
</html>
```

Slika 4.6: *HTML* šablon sa umetnutim *Python* delovima

Sada, u glavnom *Python* fajlu ovaj obrac možemo koristiti pomoću funkcije `render` za generisanje stranice (Slika 4.7).

```
class MainHandler(tornado.web.RequestHandler):
  def get(self):
    items = ["Artikal 1", "Artikal 2", "Artikal 3"]
    self.render("index.html", title="Moj naslov", items=items)
```

Slika 4.7: Generišemo šablon sa stvarnim parametrima – naslovom i stavkama liste

Naredbe kontrole toka uokvriene su sa `{%` i `%}`, dok se izrazi nalaze između dva para vitičastih zagrada - `{{` i `}}`. Naredbe kontrole toka sintaksno se ne razlikuju od svojih *Python* parnjaka. Podržane su narebe `if`, `for`, `while` i `try` i sve one se završavaju linijom `{% end %}`. S druge strane, izrazi mogu biti bilo koji smisleni *Python* izrazi, uključujući i pozive funkcija. Navedeni izrazi doslovno se kopiraju u *Python* funkciju koja implementira šablon, te ne postoje nikakva ograničenja toga šta se može napisati. Sa pozitivne strane, to pruža veću fleksibilnost programeru u odnosu na druge, strožije, *Python* sisteme šablona.

Nažalost, postoji i negativna strana a to je da pisanjem nasumičnih stvari unutar šablona dobijamo i nasumične *Python* greške.

Moduli korisničkog interfejsa

Prateći tipičnu *Python*-ovu filozofiju o neponavljanju koda bez potrebe, uvodimo module u delove naših šablona. Moduli su ponovo upotrebive komponenete kojima označavamo, stilizujemo i dodeljujemo ponašanja šablonima. Elementi koje definišemo u njima obično se koriste u više šablona, ili na više mesta u jednom šablonu. Sami po sebi, moduli predstavljaju *Python* klase koje nasleđuju *Tornado*-vu `UIModule` klasu i definišu `render` metod. Kada šablon referiše na modul koristeći sintaksu `{% Module Foo(...) %}`, *Tornado* poziva `render` metod koji vraća nisku koja se ugrađuje na mesto modula u šablonu (Slika 4.8).

```
<html>
  <head><title>UI Module Example</title></head>
  <body>
    {% module Hello() %}
  </body>

</html>
```

Slika 4.8: Uključujemo modul `Hello`

Kako bismo dovršili ugrađivanje u prethodnom primeru, modul moramo deklarisati u podešavanjima aplikacije i za to koristimo `ui_modules` parametar koji očekuje rečnik gde je ključ naziv modula a vrednost – klasa koja ga generiše (Slika 4.9).

```
class MainHandler(tornado.web.RequestHandler):
  def get(self):
    self.render('hello.html')

class HelloModule(tornado.web.UIModule):
  def render(self):
    return '<h1> Hello, world</h1>'


def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ],
    ui_modules = {'Hello' : HelloModule}
    )
```

Slika 4.9: Modul generiše jednu jednostavnu veb stranu

Vrlo često, korisnije je da modul referiše na šablon koji će se generisati, nego da generiše stranicu unapred zadatom niskom. Dobar slučaj upotrebe ovog načina je kada modul treba da iterira po rezultatima upitna nda bazom podataka (ili nekog upita aplikativnom programabilnom interfejsu), ispisujući uvek jedan formular sa različitim podacima. Recimo da od baze podataka dobijamo tražene informacije o knjigama – naziv knjige i sliku početne strane. Tada, u kodu koji

je zadužen za tu stranicu, kroz petlju bismo pozviali modul, sa prosleđivanim knjigama (Slika 4.10).

```
<html>
  <head><title>Knjige</title></head>
  <body>
      {% for book in books %}
        {% module Book(book) %}
      {% end %}
  </body>

</html>
```

Slika 4.10: Pozivamo modul sa poslatim parametrom

U ovom primeru, sam modul (Slika 4.12) prosto generiše *HTML* kod na osnovu šablona (slika 4.11).

```
<div>
    <h3 > {{ book["title"] }} </h3>
    <img src = "{{ book["image"] }}"/>
</div>
```

Slika 4.11: Koristeći sintaksu dvostrukih vitičastih zagrada pristupamo poslatim parametrima

Glavna razlika je što sada u `render` metodu klase koja nasleđuje `UIModule` nemamo generički string koji smo ranije samo ispisivali, već pozivamo `render_string`.

```
class BookModule(tornado.web.UIModule):
  def render(self, book):
    return self.render_string(
      "modules/book.html",
      book=book
    )
  def embedded_css(self):
      return ".book { margin-bottom: 1em; }"
```

Slika 4.12: `render_string` generiše šablon sa datim parametrima.

Za dodatnu fleksibilnost, *Tornado* dozvoljava ugrađivanje *CSS* i *JavaScript* fajlova u module koristeći funckije `css_files()` i `javascript_files()` čiji rezultat treba da bude lokacija navedenih fajlova na disku ili `embedded_css()` i `embedded_javascript()` čije su povratne vrednosti niske koje će biti ugrađene u module. Na slici 4.12 navodimo primer takvog stilizovanja stranice.

4.5 Autentifikacija i bezbednost

4.5.1 Kolačići

Postavljanje privatnih podataka na mreži nepobitno povlači sa sobom i pitanje njihove bezbednosti, što neretko može imati cenu u čitkosti i složenosti koda. Srećom, radni okvir *Tornado* je dizajniran uzimajući u obzir brojne bezbednosne karakteristike omogućavajući mu zaštitu protiv nekih "popularnih" vidova napada. Označeni kolačići (engl. *signed cookies*) sprečavaju prikriveno menjanje stanja korisnika, od strane zlonamernog koda, u njegovom pregledaču. Uz to, kolačići mogu biti upoređivani sa parametrima *HTTP* zahteva radi prevencije *krivotvorenja zahteva sa više strana* (engl. *cross-site request forgery - CSRF* ili *XSRF*). Na slici 4.13 vidimo primer postavljanja kolačića korisničkog pregledača.

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        if not self.get_cookie("kolacic"):
            self.set_cookie("kolacic", "vrednost")
            self.write("Kolacic nije bio postavljen ranije.")
        else:
            self.write("Kolacic je bio postavljen.")
```

Slika 4.13: Metodi za postavljanje i dobijanje vrednosti kolačića

Ovakvi kolačići, sami po sebi, nisu bezbedni i klijenti ih mogu menjati. Za nešto pouzdanije potrebe (tipa identifikacije trenutno ulogovanog korisnika) kolačići se moraju označiti radi prevencije falsifikovanja. Prvo, treba postaviti tajni ključ u aplikacijskim podešavanjima.

```
application = tornado.web.Application([
    (r"/", MainHandler),
], cookie_secret="__TODO:_UNETI_NASUMIČNU_VREDNOST_OVDE__")
```

Slika 4.14: Postavljanje `cookie_secret` parametra

Ova vrednost bi trebala da bude jedinstvena nasumično odabrana niska. Za dobijanje takve vrednosti možemo iskoristiti 128-bitni univerzalni jedinstveni identifikator (engl. *universally unique identifier*) dobijen pomoću *Python*-ovog `uuid4()` metoda na sledeći način:

```
base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes
```

Zatim, prethodni primer možemo prepraviti tako da radi sa označenim kolačićima (Slika 4.15).

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        if not self.get_secure_cookie("kolacic"):
            self.set_secure_cookie("kolacic", "vrednost")
            self.write("Kolacic nije bio postavljen ranije.")
        else:
            self.write("Kolacic je bio postavljen.")
```

Slika 4.15: Metodi za postavljanje i dobijanje vrednosti označenih kolačića

Uz navedenu kodiranu vrednost, *Tornado* sadržaju kolačića dodaje i vremenski žig (engl. *timestamp*) i *HMAC*[22] potpis radi dodatne bezbednosti. Po primljenom zahtevu, čita se kolačić i ukoliko je vremenski žig previše star (ili dolazi iz budućnosti) ili se potpisi ne slažu, *Tornado* pretpostavlja da je kolačić nasilno menjan, te će `get_secure_cookie` vratiti `None`.

Ovaj vid kolačića ipak nije otporan na zlonamerne napade. Uzurpator može, koristeći skripte ili *dodatne komponente pregledača* (engl. *plug-ins*) da presretne nešifrovane podatke. Treba imati u vidu da vrednosti kolačića nisu šifrovani, već su označeni. Zloćudni programi mogu da čitaju sačuvane kolačiće i prenose njihove podatke određenim serverima, ili da šalju sopstvene zahteve vraćajući ih aplikaciji ne menjavši ih prethodno. Zato, trebalo bi izbegavati skladištenje osetljivih korisničkih podataka u kolačiću.

4.5.2 Korisnička autentifikacija

Trenutno prijavljeni korisnik, dostupan je svakom rukovaocu kao `self.current_user` i svakom šablonu kao `current_user`. Podrazumevano, `current_user` je `None`. Za implementaciju autentifikacije potrebno je nadjačati `get_current_user()` metod, u odgovarajućim rukovaocima, koji određuje trenutno prijavljenog korisnika (Slika 4.16).

```
class MyHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")
```

Slika 4.16: Primer određivanja trenutnog korisnika

Možemo zahtevati od korisnika da se prijavi na sistem kako bi dobio privilegiju pristupa nekoj stranici. Dovoljno je postaviti dekorator `@tornado.web.authenticated` iznad metoda koji generiše tu stranicu. Ukoliko se radi o nepoznatom korisniku, on se automatski preusmerava na stranicu datu u podešavanjima aplikacije kroz `login_url`. Ukoliko se ovaj dekorator nalazi iznad nekog `post()` metoda i korisnik nije prijavljen, zahtev će rezultovati *403 - Forbidden* statusnim kodom.

[22] *HMAC*, odnosno *Hash-based Message Authentication Code*, je kod dobijen kriptografskim heš funkcijama i tajnim kriptografskim ključem. Prvi put se pominje u članku iz 1996. godine, dok u februaru 1997. godine izlazi i RFC [35].

Autentifikacija treće strane

Tornado-v `tornado.auth` modul implementira autentifikacijske i autorizacijske protokole koristeći više popularnih platformi kao što su *Google/Gmail*, *Facebook* i *Twitter*. Pored metoda za registrovanje i upis korisnika, tu su i metodi odobrenja pristupa nekim servisima (gde je to izvodljivo), tako da je moguće preuzimanje korisnikovog adresara ili objavljivanje *Twitter* poruka u korisnikovo ime (Slika 4.17).

```
class TwitterLoginHandler(tornado.web.RequestHandler,
                          tornado.auth.TwitterMixin):
  async def get(self):
    if self.get_argument("oauth_token", None):
      korisnik = await self.get_authenticated_user()
    else:
      await self.authorize_redirect()
```

Slika 4.17: Primer integrisanja *Twitter* korisnika u aplikaciju

4.5.3 Krivotvorenja zahteva sa više strana

Ovaj vid napada je tipičan za personalizovane veb aplikacije jer se oslanja na poverenje koje prijavljeni korisnik uživa na serverskoj strani. Žrtva (klijent) nesvesno šalje zlonamerno napravljen veb zahtev aplikaciji kod koje ima privilegovani pristup. Ovaj zahtev uključuje URL parametre, kolačiće i ostale podatke koje server tretira kao običan klijentski zahtev. Generalno prihvaćeno rešenje prevencije ovakvih prevara jeste da se svakom korisniku dodeli nepredvidiva vrednost koja se zatim koristi kao dodatni argument pri podnošenju svakog zahteva. Ukoliko se vrednost kolačića i vrednost parametra u poslatom formularu ne poklapaju, vrlo je verovatno da je formular lažan. Radni okvir *Tornado* dolazi sa ugrađenom zaštitom protiv ovakvih napada. Za njeno korišćenje, prethodno treba u podešavanjima aplikacije vrednost `xsrf_cookies` postaviti na `True`. Kda je ovo urađeno, kolačić će se dodeljivati korisnicima i svi *POST*, *PUT* i *DELETE* zahtevi koji ne sadrže tačnu vrednost će biti odbijeni. Za uspešno slanje zahteva potrebno je još opremiti sve formulare koji šalju *POST* zahteve. Ovo je moguće integracijom specijalnog modula `xsrf_form_html()` u formular, dostupnog u svim modulima (Slika 4.18).
Ako se zahtevi izvršavaju pomoću *AJAX*-a, onda treba i *JavaScript* kod urediti tako da uključuje `_xsrf` vrednosti.

```
<form action="/new_message" method="post">
  {% module xsrf_form_html() %}
  <input type="text" name="message"/>
  <input type="submit" value="Post"/>
</form>
```

Slika 4.18: Uključivanje modula u formular za slanje

Ukoliko sada pogledamo *izvorni kod* (engl. *source code*) kroz alate veb pregledača videćemo nešto poput sledećeg:

Slika 4.19: Prikaz izvornog koda prethodnog primera

Poglavlje 5

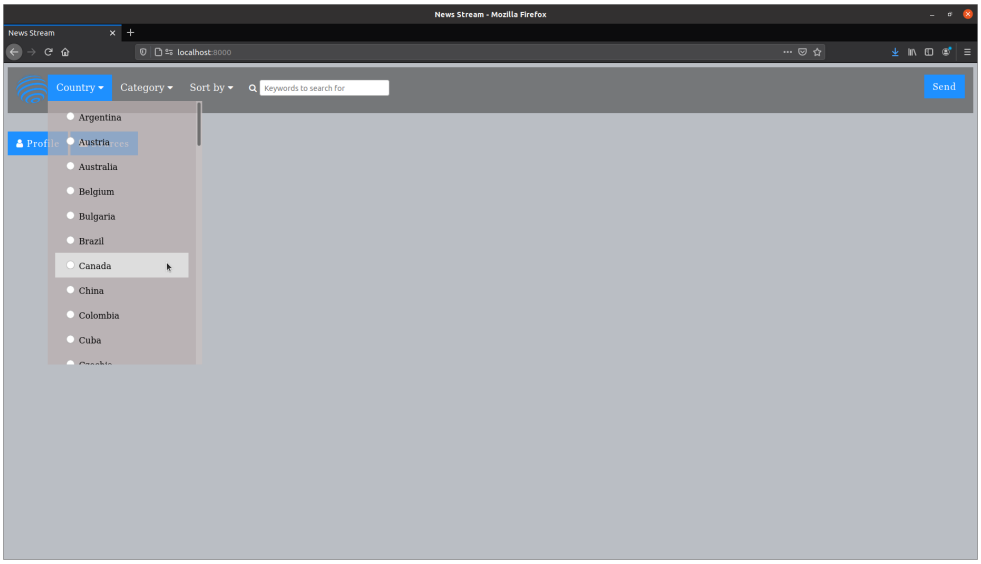
Aplikacija

U narednim sekcijama, pokazaćemo funkcionalnosti radnog okvira *Tornado* na konkrentom primeru jedne veb aplikacije.

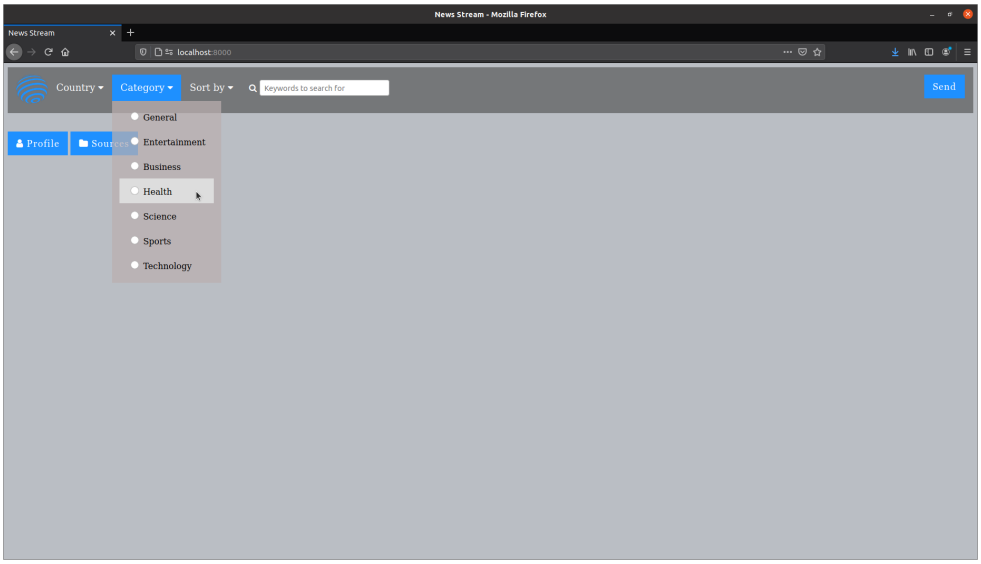
5.1 Opis aplikacije

Cilj aplikacije je prikazivanje vesti i informacija iz zemlje i sveta, uz dodatne mogućnosti poput čuvanja ili brisanja vesti za autentifikovane korisnike. Dakle, u aplikaciju je integrisan i sistem registracije novih korisnika, ali i prijavljivanja i odjavljivanja već postojećih. Aplikacija je otvorenog koda i dostupna je na *GitHub* repozitorijumu na adresi `https://github.com/st3vo7/NewsStream`. Sama aplikacija sastoji se od klijentskog dela napisanog koristeći *Torando*-ve šablone, uz prateće funkcionalnosti omogućene korišćenjem programskog jezika *JavaScript*, i to pre svega biblioteke *jQuery* i *AJAX* tehnika, dok je serverski deo urađen koristeći biblioteku i radni okvir *Torando*. Aplikacija potražuje podatke od eksternog aplikativnog programabilnog interfesja *NEWSAPI* [31], koje potom raspakuje, uređuje i prikazuje krajnjim korisnicima.

Korisnik na početnom ekranu, sa liste padajućeg menija, vrši odabir države za koju želi da dobija vesti (Slika 5.1) i uz to može odabrati jednu od kategorija: **Opšte**, **Zabava**, **Biznis**, **Zdravlje**, **Nauka**, **Sport** i **Tehnologija** (Slika 5.2). Napomenimo i da izbor kategorije nije obavezan i u tom slučaju prikazivaće se opšte vesti, dokle god korisnik ne promeni podrazumevanu vrednost.

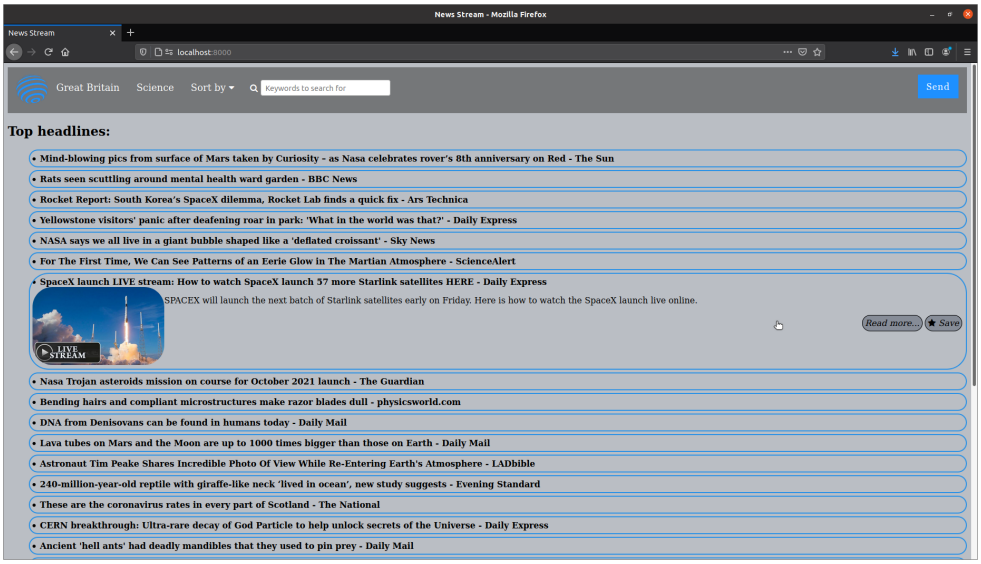


Slika 5.1: Korisnik bira jednu od 54 ponuđene države



Slika 5.2: Korisnik bira neku od ponuđenih kategorija

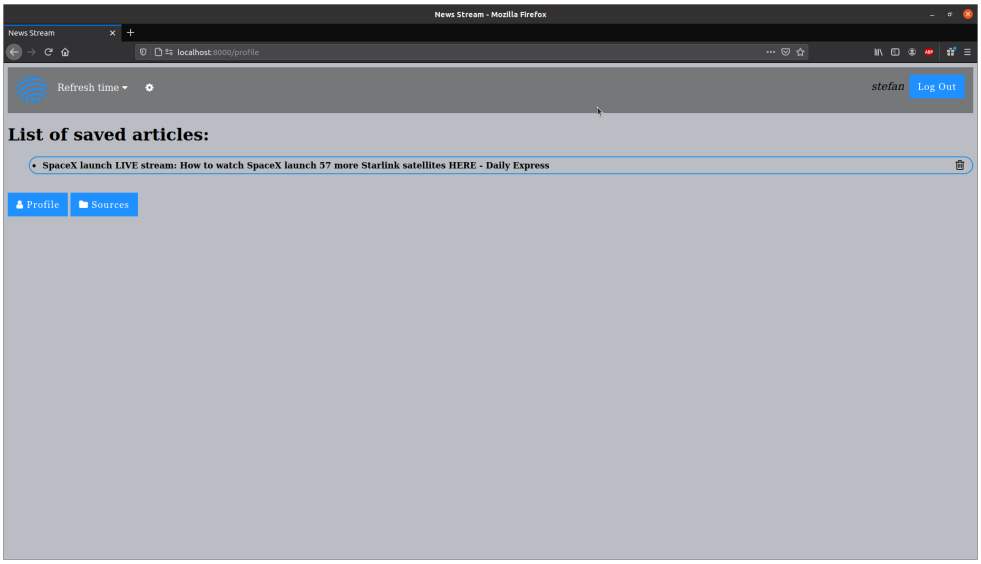
Dalje, korisnik dobijene vesti može da sortira po datumu rastuće i opadajuće (podrazumevano). Osim toga, ukoliko se kao parametar pošalje i ključna reč, korisniku će se prikazati samo sadržaj koji sadrži tu reč ili frazu. Što se tiče samog sadržaja, korisniku se prikazuju naslovi koji se klikom "proširuju" i korisnik tada može da vidi prikladnu sliku i kratak opis vesti, ukoliko postoji (Slika 5.3).



Slika 5.3: Najnoviji naslovi za odabrane parametre

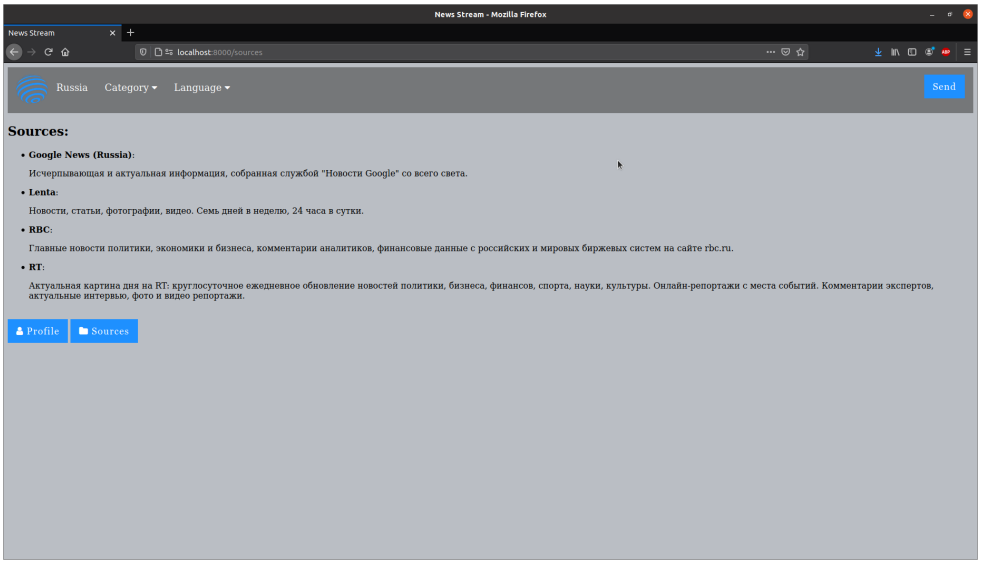
Uz to, korisnik može da, klikom na odgovarajuće dugme, pročita kompletnu vest na matičnom sajtu ili da sačuva vest na svom nalogu. Inicirajući komandu čuvanja, neprijavljeni korisnik, vodi se na stranicu za prijavu, odnosno registraciju korisnika. Na taj način korisnik je motivisan da ostvari pravo korišćenja svih funkcionalnosti ove veb aplikacije.

Na profilnoj stranici korisnika nalaze se sačuvane vesti, modul za promenu vremena čekanja na potragu novih vesti, kao i modul za promenu sačuvane lozinke (Slika 5.4).



Slika 5.4: Izgled korisnikove profilne stranice

Osim vesti, klijent može i da potražuje izvore koji objavljuju vesti (Slika 5.5). Od dostupnih parametara, tu su država iz koje dolazi izvor, kategorija vesti koje objavljuje, kao i jezik na kome se vesti objavljuju.

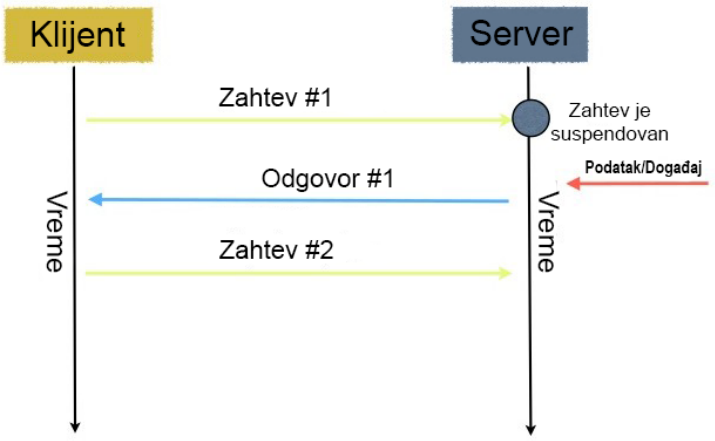


Slika 5.5: Izgled stranice sa izvorima

5.2 Dugotrajno anketiranje ili Veb utičnice

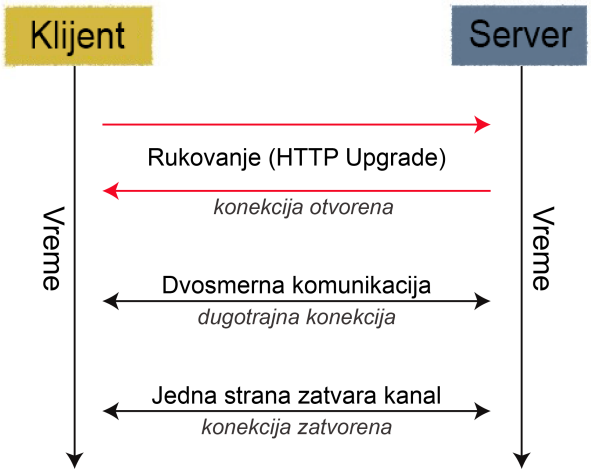
Dugotrajno anketiranje predstavlja tehniku veb komunikacije u kojoj se klijent obraća serveru potražujući podatke, a server bira da ostavlja konekciju otvorenom, dostavljajući tražene podatke kada oni postanu dostupni, ili dođe do isteka *vremenskog ograničenja* (engl. *timeout*) nakon čega zatvara konekciju. S druge strane, po prijemu novih podataka, klijentov pregledač odmah šalje novi zahtev sa istim parametrima inicirajući novu rundu razmene poruka (Slika 5.6).

Sama tehnika predstavlja poboljšanje obične, *HTTP* tehnike anketiranja u kojoj se zahtevi serveru šalju u unapred određenim vremenskim intervalima, što može biti zgodno ukoliko imamo predstavu o tome koliko često novi podaci pristižu serveru. Ukoliko to nije slučaj, ovakva tehnika nije adekvatna, te je potrebno poboljšanje, koje se postiže upotrebom tehnike dugotrajnog anketiranja. Primetimo da u slučaju velikog broja razmene poruka (što se, u suštini, svodi na učestalo dobijanje "svežih" podataka na serverskoj strani), dugotrajno anketiranje ne predstavlja nikakvo poboljšanje.



Slika 5.6: Komunikacija između klijentovog pregledača i servera korišćenjem dugotrajnog anketiranja

Veb utičnica označava protokol koji obezbeđuje komunikaciju *dvosmernim kanalima* (engl. *full-duplex*) preko jedne *TCP* konekcije. Korišćenje dvosmernog kanala omogućava klijentu i serveru da razmenjuju poruke kroz jednu, dugotrajnu, konekciju (Slika 5.7).

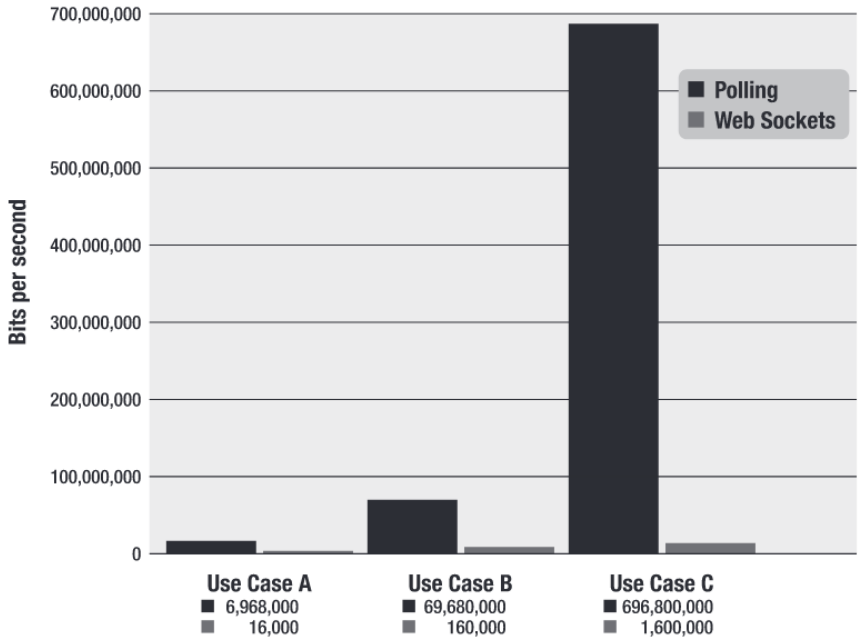


Slika 5.7: Komunikacija između klijentovog pregledača i servera korišćenjem veb utičnice

Nakon inicijalnog *rukovanja* (engl. *handshake*) u kome se klijent predstavlja serveru i obratno, ostvarujući na taj način sigurnu konekciju, komunikacija se neometano odvija dokle god neko od njih ne zatvori ovaj kanal. Iako se zasnivaju na protokolu koji nije *HTTP*, veb utičnice jesu kompatabilne sa takvim zahtevima koristeći zaglavlje `Upgrade` *HTTP* protokola.

Kao i ostale tehnologije i ove imaju svoje prednosti i mane. Pogodnosti korišćenja dugotrajnog anketiranja leže u njegovoj jednostavnosti implementacije. S obzirom na to da koriti samo *HTTP* protokol, gotovo da nije potrebna nikakva

dodatna podrška u veb pregledaču. Što se mana tiče, verovatno je najveća to što porastom broja zahteva opterećenje servera drastično raste, kao i slanje bespotrebnog mrežnog saobraćaja (Slika 5.8). Dodatno, treba obratiti pažnju na pravilan redosled stizanja poruka pri konkurentnim zahtevima.



Slika 5.8: Poređenje mrežnog viška između dugotrajnog anketiranja i veb utičnica

Izvor: [32] str. 144, slika 6-3

Što se tiče veb utičnica, nakon otvaranja jedne dugotrajne konekcije, nema potrebe za slanjem zaglavlja, te će i ukupni bespotrebni mrežni saobraćaj biti smanjen. Negativna strana je to što je veb utičnicama potrebna dodatna podrška od strane veb pregledača. Sa rastom popularnosti ovog vida komunikacije, pregledači u sve većem broju dolaze sa ugrađenom podrškom veb utičnicama, te je ovaj problem sve manje relevantan.

5.2.1 Podrška u radnom okviru Tornado

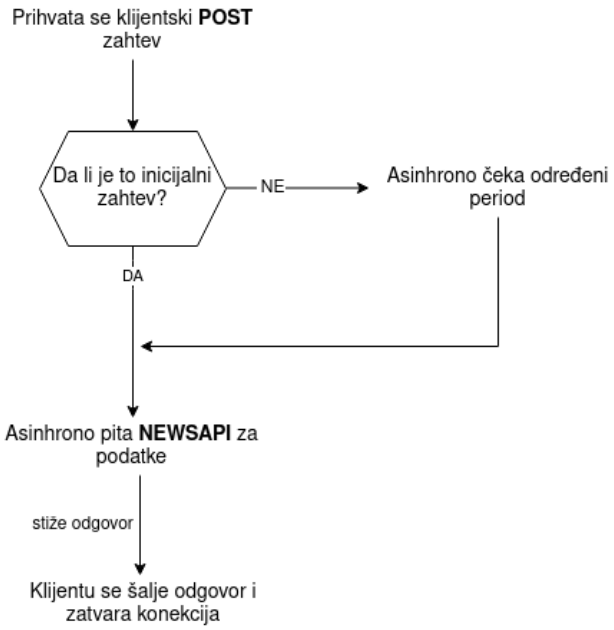
Kao što je već navedeno, tehnike dugotrajnog anketiranja baziraju se nad običnim *HTTP* zahtevima koji se kroz radni okvir *Tornado* mogu asinhrono organizovati bez previše komplikacija. Veb utičnice su takođe podržani kroz `tornado.websocket` biblioteku. Međutim, kako se sami ne baziraju na *HTTP* konekcijama, to i većina *HTTP* resursa nije dostupna. Jedini dostupni komunikacijski metodi su `write_message()`, `ping()` i `close()`. Uz to, klase rukovalaca zahteva trebalo bi da implementiraju metod `open()`, umesto klasičnih `get()` ili `post()`.

Radeći na aplikaciji, tendencija je bila da se ispoštuju svi standardi u pisanju veb aplikacija, kao i da se očuva jednostavnost i čitljivost koda, uz dobru ilustraciju načina rada u *Tornado* okruženju. Upravo zbog svoje jednostavnosti,

dobro poznatog ponašanja i činjenice da se promene u podacima ne dešavaju na nivou sekunda (ili desetetog dela sekunda), te momentalna izmena na korisničkioj strani nije nužna, ovde je odabran pristup dugotrajnog anketiranja.

5.3 Način rada aplikacije

Kao što smo već naveli, korisnik potražuje podatke od servera, koji upit prosleđuje eksternom interfejsu. Kada korisnik dobije tražene vesti, inicira se novi zahtev serveru, sa istim parametrima. Naravno, server neće odmah upitati interfejs za vesti, već će klijentov zahtev biti "uspavan" određeni vremenski period (prodrazumevano, to je 10 minuta), nakon čega će se izvršiti upit i klijentu će se proslediti rezultat. Uspavljivanje klijentovog zahteva je postavljeno zarad racionalnijeg slanja upita eksternom interfejsu. Podaci (vesti) koje stižu ne generišu se brže on reda veličine minuta, te je slanje upita na svakih nekoliko sekundi bespotrebno. Dalje, klijent upoređuje dobijene vesti sa onima koje su na stranici, i ažurira listu naslova dodajući nove, ako postoje (Slika 5.9). Ovaj postupak se ponavlja dokle god je korisnik na početnoj stanici, odnosno, dok ne promeni parametre, kada kreće novi ciklus razmene poruka između klijenta i servera.



Slika 5.9: Dijagram rada servera prilikom obrade klijentskog zahteva

Ispratićemo ovaj dijagram i *Python* kodom (Slika 5.10).

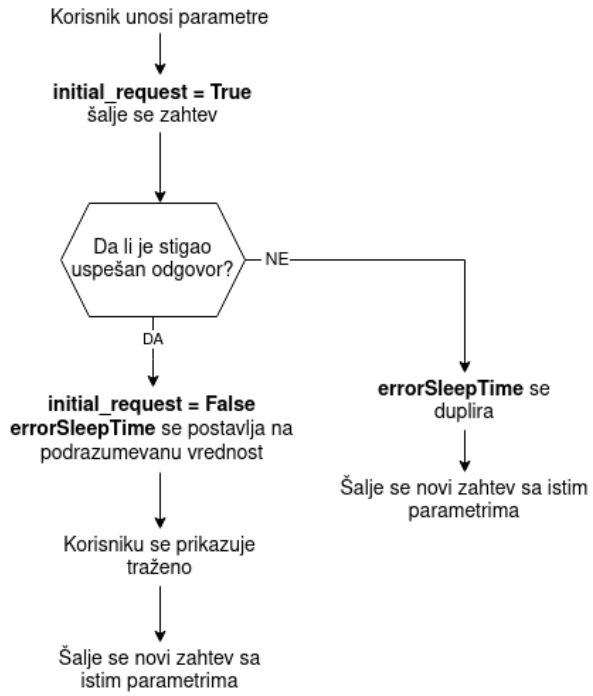
```
dic_data = tornado.escape.json_decode(self.request.body)
country = dic_data['country']
category = dic_data['category']
initial_request = dic_data['initial_request']
keyword = dic_data['keyword'] if 'keyword' in dic_data else ''

if not initial_request:
        timer = 600

        if self.current_user is not None:
                v1 = await do_find_one(collection, self.current_user)
                timer = v1['timer']

        self.sleeping_client = asyncio.sleep(timer)
        await self.sleeping_client

url = ('https://newsapi.org/v2/top-headlines?'
        'country=' + country + '&'
        'category=' + category + '&'
        'pageSize=100&'
        'q=' + keyword + '&'
        'apiKey=17060bbc869845deb9246555cd6f8e5d')

http_client = tornado.httpclient.AsyncHTTPClient()
try:
        response1 = await http_client.fetch(url)
except Exception as e:
        print("Dogodila se greska: %s" % e)

my_data = tornado.escape.json_decode(response1.body)
self.write(json.dumps({'sent': my_data, 'country': a,
                       'category': b, 'keyword': c}))
```

Slika 5.10: Prijem zahteva, potraznja podataka i slanje klijentu

Dakle, nakon što detektujemo zahtev i "izvučemo" parametre koje nam je klijent poslao, sledi nam moguće čekanje zavisno od toga da li je to inicijalni zahtev ili ne. U slučaju da nije, opet, imamo više slučajeva. Podrazumevana vrednost čekanja je 600 sekundi, ukoliko naš klijent nije ulogovan. Ukoliko jeste, on ima dozvolu da promeni podrazumevano vreme čekanja.[23] Koristeći funkcionalnosti biblioteke `asyncio` "uspavljivanje" je asinhrono, te je server u stanju da prihvata zahteve drugih klijenata u međuvremenu. Po isteku vremena čekanja, generiše se veb-adresa na kojoj se nalaze odgovarajući podaci. Pomoću `AsyncHTTPClient()` sada vršimo jedan asinhroni upit i po njegovom završetku šaljemo klijentu podatake.

Analogno prethodnom, daćemo i prikaz rada klijentovog pregledača (Slika 5.11). Na dijagramu postoji par delova čiju funkcionalnost treba detaljnije objasniti. Naime, ukoliko server iz nekog razloga nije dostupan, odgovor će stići odmah kao *500 Internal Server Error*, te će, podrazumevano, klijentov pregledač odmah poslati novi zahtev – dobijajući opet isti odgovor itd. Kako bismo sprečili ovo uzaludno slanje zahteva, uvodimo promenljivu `errorSleepTime` kojom postepeno usporavamo broj slanja zahteva (sada neaktivnom) serveru.

[23] Mogući izbor vrednosti je: 5, 10, 15, 30 i 60 minuta.



Slika 5.11: Dijagram rada klijentovog pregledača

Druga promenljiva koju uvodimo na klijentu je `initial_request` koja razlučuje da li je je u pitanju inicijalni zahtev podnet od strane korisnika, ili je u pitanju automatski (što je karatkeristika dugotrajnog anketiranja. Naime, ukoliko to zaista jeste inicijalni zahtev, pregledač će sada znati da treba da izbriše sve što se do tada nalazilo na stranici i da ispiše nove vesti, dok u suprnotnom, iz dobijenog sadržaja treba izdvojiti nove vesti i dodati ih na listu postojećih (Slika 5.12). Isti parametar koristi i server za odluku o suspendovanju klijenta.

```
function data_ok(data) {

    errorSleepTime = 500;

    obj = JSON.parse(data);
    articles = obj.sent.articles;

    if (sending_parameters.initial_request) {
        /*
        ...
        kod za obradu i prikaz članaka u slučaju inicijalnog zahteva
        ...
        */
    }
    else {
        /*
        ...
        kod za dodavanje novih članaka na listu starih
        ...
        */
    }
    sending_parameters.initial_request = false;
    window.setTimeout(post_ajax.bind(null, 'http://localhost:8000',
                                      JSON.stringify(sending_parameters),
                                      data_ok, data_not_ok), 0);
}
```

Slika 5.12: Slučaj uspešnog pristizanja podataka sa servera

5.4 NewsAPI

Kao što smo već naveli, podaci koji se obrađuju i prikazuju korisnicima dobijeni su korišćenjem eksternog upita aplikativnom programabilnom interfejsu *NewsAPI*. U pitanju je jedan jednostavan interfejs, koji na upit daje rezultat *JSON* formata. Postoje tri različite krajnje tačke (engl. *endpoints*) – `top-headlines`, `everything` i `sources` sa kojih se mogu dobijati podaci. Prve dve rezultuju vestima, dok je treća rezervisana za dostupne izvore koji se mogu dobiti. U našoj aplikaciji korištene su prva i poslednja. Registracijom na https://newsapi.org/ dobija se jedinstveni privatni ključ koji se šalje kao parametar pri svakom obraćanju interfejsu. Na primer, da bismo dobili trenutno najnovije relevantne vesti u Sjedinjenim Američkim Državama, možemo, koristeći alat `curl`, uputiti zahtev na sledeći način: `curl http://newsapi.org/v2/top-headlines -G -d country=us -d apiKey=17060bbc869845deb9246555cd6f8e5d`. Što se tiče objekta koji se dobija, u pitanju je rečnik sa sledećim poljima:

Polje	*Opis polja*
`status`	Sadrži podatak o tome da li je zahtev bio uspešan ili ne.
`totalResults`	Ukupan broj rezultata dostupnih u zahtevu
`articles`	Rezultati zahteva
`source`	Identifikator i naziv izvora
`author`	Autor članka
`title`	Naslov članka
`description`	Opis ili kratak deo članka
`url`	Veb adresa ka članku.
`urlToImage`	Veb adresa ka naslovnoj slici.
`publishedAt`	Datum i vreme objave.
`content`	Neformatiran sadržaj članka, skraćen na 200 karaktera.

Ukupan broj rezultata predstavlja broj dobijenih artikala. Polje rezultati zahteva je zapravo niz rečnika od kojih svaki ima ključeve *author*, *content*, *description*, *publishedAt*, *source*, *title*, *url* i *urlToImage*, dok se vrednosti razlikuju (jer su i članci različiti).

Ukoliko ključ nije poslat korektno ili nije validan, zahtev će rezultovati greškom `401 HTTP error - Unauthorized`.

5.4.1 Python podrška

Postoji neoficijelna klijentska biblioteka koja integriše *NewsAPI* u *Python* kod. Instalira se komandom *Python*-ovog sistema za upravljanje paketima `pip3 install newsapi-python` i potom se može koristiti slično ostalim bibliotekama (Slika 5.13).

```
from newsapi import NewsApiClient

na = NewsApiClient(api_key='17060bbc869845deb9246555cd6f8e5d')
th = na.get_top_headlines(q='bitcoin',
                          category='business',
                          country='us')

print(th)
```

Slika 5.13: Primer jednostavnog upita

U želji da koristimo asinhroni pristup u aplikaciji, a imajući u vidu da nad metodama za dobijanje sadržaja ove biblioteke ne mogu primeniti asinhroni pozivi, odlučili odlučili smo se da u aplikaciji ne koristimo ovu biblioteku, vec da koristimo Tornadov intefejs *Tornado*-v interfejs `AsyncHTTPClient()` uz "ručno" generisanje adresa.

5.5 Upravljanje bazama podataka

Počevši od verzije *3.0*, *Tornado* nema sopstvenu podršku za upravljanje bazama podataka, te se oslanja na spoljašnje pakete. Za nešto starije verzije dostupna je *Torndb* [33] biblioteka (koja je zapravo omotač oko *MySQLdb*) no kako više nije aktivno održavana i nekompatabilna je sa *Python 3*, nema preteranog smisla koristiti je.

Na zvaničnoj stranici nalazi se lista biblioteka čije se funkcionalnosti mogu koristiti u razvoju *Tornado* aplikacija. Od podrške sistemima za upravljanje bazama podataka izdvojićemo *TorMySQL* kao upravljač (engl. *driver*) za *PyMySQL* [24] [36], kao i *Psycopg* i *Aiopg* biblioteke za *PostgreSQL*. Od biblioteka za nerelacione baze podataka navešćemo *Motor* koja predstavlja asinhronu sponu između *Python* koda i *MongoDB* sistema.

5.5.1 Nerelacione baze podataka

Za razliku od relacionih baza podataka, koje svoje podatke čuvaju u relacionim tabelama, *nerelacione* (engl. *non-relational*) baze čuvaju svoje podatke u drugim formatima. Neki od najznačajnijih tipova nerelacionih baza podataka su:

Tip dokumenta: Ovakve baze podataka čuvaju podatke u dokumentima formata sličnog *JSON*-u. Svaki dokument sadrži parove ključeva i vrednosti. Vrednosti mogu biti različitih tipova poput niski, brojeva, nizova i sl. *MongoDB* je primer sistema koji radi sa ovakvim tipom baze.

Tip ključ-vrednost: Predstavlja jednostavniji tip baze, gde se stavke takođe sastoje od ključeva i vrednosti i predstavljaju dobar izbor pri skladištenju velikog broja podataka gde nisu potrebni komplikovani upiti za njihovo izdvajanje. Jedan od primera upotrebe je pri čuvanju korisničkih izbora ili keširanju. *Redis* i *DynamoDB* su primeri ovakvog tipa baze.

Tip proširene kolone: Podaci se čuvaju u tabelama, redovima i dinamičkim kolonama. Pružena je dodatna fleksibilnost (u poređenju sa relacionim tabelama) jer redovi ne moraju imati identične kolone. Koriste se za skladištenje podataka *interneta stvari* (engl. *Internet of Things - IOT*). Primeri ovog tipa baze su *Cassandra* i *HBase*.

[24] *PyMySQL* je biblioteka za manipulaciju nad relacionim bazama podataka kroz *Python*. Za uspešnu integraciju, potrebno je imati instaliran neki sistem za upravljanje bazama podataka poput *MySQL* ili *MariaDB*. Zarad potpune iskoristivosti asinhronog pristupa, neophodan je asinhroni upravljač koji neće blokirati čitav program prilikom upita ka bazi podataka, te je preporuka koristiti *TorMySQL*.

Grafovski tip: Baze ovog tipa čuvaju podatke u *čvorovima* (engl. *nodes*) i *ivicama* (engl. *edges*). Na primer, ako se u čvorovima čuvaju podaci o mestima ili ljudima, onda ivice predstavljaju podatke o vezama između tih mesta, odnosno ljudi. Tipičan primer upotrebe su socijalne mreže, i sistemi preporučivanja (engl. *recommendation engines*), dok su primeri ovakvih baza podataka *Neo4j* i *JanusGraph*.

Pogrešno je mišljenje da nerelacione baze podataka ne mogu da dobro dokumentuju odnose između podataka. Mogu, samo to rade na drugačiji način u odnosu na relacione baze podataka. Pogledajmo sledeći primer. Na slici 5.14 vidimo relacionu tabelu koja opisuje korisnike date šifrom, imenom, prezimenom i brojem godina.

KORISNICI			
primarni_ključ	ime	prezime	godine
0	Petar	Petrović	24
1	Jovana	Jovanović	28

Slika 5.14: Relaciona tabela korisnika

Ukoliko bismo hteli ovim korisnicima dodeliti određene atribute (recimo da želimo evidentirati i njihovu decu) napravili bismo novu tabelu koja bi obavezno sadržala identifikatore prve tabele (Slika 5.15).

DECA				
primarni_ključ	ime	prezime	godine	id_roditelja
0	Marko	Petrović	12	0
1	Jelena	Petrović	7	0
2	Janko	Jovanović	9	1
3	Radomir	Jovanović	14	1
4	Anastasija	Jovanović	13	1

Slika 5.15: Relaciona tabela dece korisnika

Kako se u nerelacionim bazama podataka[25] podaci ne čuvaju u tabelama, ovakve veze između entiteta bismo implementirali na nešto drugačiji način (Slika 5.16).

[25]Konkretno, mslimo na baze koje podatke čuvaju u formi dokumenata.

```
KORISNICI = [
                {"primarni_ključ": 0,
                "ime": "Petar",
                "prezime": "Petrović",
                "godine": 24,
                "deca": [
                                {"ime": "Marko",
                                "prezime": "Petrović",
                                "godine": 12
                                },
                                {"ime": "Jelena",
                                "prezime": "Petrović",
                                "godine": 7
                                }
                        ]
                },

                {"primarni_ključ": 1,
                "ime": "Jovana",
                "prezime": "Jovanović",
                "godine": 28,
                "deca": [
                                {"ime": "Janko",
                                "prezime": "Jovanović",
                                "godine": 9
                                },
                                {"ime": "Radomir",
                                "prezime": "Jovanović",
                                "godine": 14
                                },
                                {"ime": "Anastasija",
                                "prezime": "Jovanović",
                                "godine": 13
                                }
                        ]
                }
    ]
```

Slika 5.16: Dokument korisnika i njihove dece

Primetimo da smo do sada navodili samo odnos `one-to-many`, tj. u bazi su podaci o jednom roditelju koji može imati više dece. Odnos `one-to-one` (jedan roditelj – jedno dete) ne bi menjao ovakvo modelovanje. Priča se malo komplikuje uvođenjem odnosa `many-to-many`, odnosno, modelovanjem baze u kojoj za jedno dete postoji više roditelja. U opštem slučaju, ovakav odnos se modeluje dodavanjem posebne tabele odnosa koja sadrži parove primarnih ključeva tabele roditelja i tabele dece.

U slučaju neralacione baze koja svoje podatke čuva u formi dokumenata, situacija je nešto komplikovanija. Dosadašnji pristup *ugrađivanja* nije primenljiv, već se mora preći na sistem *referisanja*. U jedan dokument se smeštaju podaci o roditeljima, dok su podaci o deci sada poseban dokument koji, među ostalim podacima, sadrži i listu parova identifikatora svojih roditelja (npr.`id_roditelja1` i `id_roditelja2` koji adekvatnim vrednostima referišu na dokument roditelja). Analogno, kako jedan roditelj može imati više dece, on sada treba da referiše na njih – tako, njegovim podacima pridružujemo listu identifikatora njegove dece (koji se, rekli smo, nalaze kao poseban dokument).

U nastavku daćemo opis sistema za upravljanje ovakvim bazama podataka, kao i primere upotrebe.

5.5.2 MongoDB

MongoDB je sistem za upravljanje nerelacionim bazama podataka. On pripada prvoj grupi, odnosno sistemima koje svoje podatke čuvaju u formi dokumenata. Prilikom razvoja akcenat je stavljen na jednostavnost upotrebe i skalabilnost. Pored mogućnosti upravljanja bazom kroz *Linux*-ov komandni interfejs postoji podrška za više programskih jezika kao što su *Python*, *Java*, *C++*, *JavaScript* (u okviru radnog okvira *Node.JS*) i drugi.

MongoDB čuva podatke kao *BSON dokumente*. *BSON* je zapravo binarna reprezentacija *JSON* dokumenata, ali sadrži više ugrađenih tipova podataka od samog *JSON*-a. Neki od podržanih tipova podataka su `ObjectId`, `String`, `Double`, `Boolean`, `Date` i dr. Dokumenti su organizovani u *kolekcije*. Baza podataka koristi jednu ili više kolekcija dokumenata.

Postavljeno ograničenje na veličinu *BSON* dokumenta iznosi 16 megabajta. Ovo je urađeno da bi se sprečilo generisanje apsurdno velikih dokumenata čija bi pretraga koristila velike količine *RAM* memorije.

Nazivi polja (ono što su ključevi u heš mapama) su obavezno niske. Dokumenti imaju neka ograničenja na nazive polja, npr. naziv polja `_id` je rezervisan za primarni ključ. Vrednost primarnog ključa je jedinstvena, nije je moguće promeniti i može biti bilo kog tipa osim agregatnog. Ukoliko pri upisu ne navedemo eksplicitno, sistem će sam dodeliti svakom objektu jedinstveni identifikator. Takođe, nazivi polja ne mogu sadržati `null` karakter. Koristeći *tačka notaciju* (engl. *dot notation*) i ugrađenu funkciju `insertOne()`, možemo da umetnemo entitet u kolekciju kao `db.KORISNICI.insertOne({"_id":2, "ime": "Marko", "prezime": "Marković", "godine": 22, "deca": []})`.

Dalje, ažuriranje obavljamo koristeći funkcije `updateOne()`, `updateMany()` i `replaceOne()`. Na primer, komandom `db.KORISNICI.updateOne({"_id":2}, {$set:{deca: [{"ime":"pavle", "prezime": "Marković", "godine": 1}]}})` ažuriramo vrednost polja `"deca"`.

Slično ovome, operacije brisanja se iniciraju komandama `deleteOne()` i `deleteMany()`. Naredbom `db.KORISNICI.deleteMany({"prezime":"Marković"})` iniciramo akciju brisanja svih "Markovića" iz kolekcije korisnici.

5.5.3 Baza podataka aplikacije

Tokom razvoja aplikacije, pojavila se potreba za pohranjivanjem podataka dobijenih od klijenata, kao i obradom sačuvanih podataka i njihovog prikazivanja klijentu. U cilju jednostavnosti, odabran je nerelacioni pristup a kao sistem za upravljanje bazom podataka korišćen je prethodno opisani *MongoDB*. U želji da se ne izgubi asinhronosti, iskoristili smo *Motor* biblioteku kao asinhronu nadgradnju na *PyMongo*[26].

Za početak, pogledajmo samu strukturu baze (Slika 5.17).

[26] *PyMongo* distribuciju čini skup alata za interakciju sa *MongoDB* bazom podataka kroz *Python*. [37]

Slika 5.17: Baza sa trenutnim podacima

Dakle, trenutno imamo upisane podatke o jednom registrovanom korisniku. Podaci se nalaze u formi rečnika, gde su ključevi `_id`, `name`, `password`, `headlines` i `timer`, a ključ `headlines` sadrži, kao vrednost, listu rečnika sa podacima o sačuvanim vestima.

Za integraciju *Motor* biblioteke, dovoljno je napraviti jednu instancu `MotorClient`-a, sa hostom i brojem porta, nakon čega se navodi ime baze i kolekcije (Slika 5.18).

```
client = motor.motor_tornado.MotorClient('localhost', 27017)
db = client.test
collection = db.test
```

Slika 5.18: Postavljanje inicijalnih podešavanja

Ostaje još napomenuti *Tornado*-u da smo spremni za upotrebu baze podataka, što radimo koristeći rečnik podešavanja aplikacije, dodeljujući vrednost imena baze ključu `"db"`.

Na primeru sa slike 5.10 koristili smo funkciju `do_find_one` za pronalaženje vrednosti tajmera trenutno prijavljenog klijenta. Sledeći primer (Slika 5.19) prikazuje njenu definiciju. Rezultat ove funkcije je rečnik koji je prvi nađen, takav da zadovoljava unete parametre. Za smeštanje podataka u bazu koristimo ugrađenu funkciju `insert_one`. Kao i *PyMongo*, i *Motor* predstavlja dokumenta baze podataka rečnicima u *Python*-u. Ugrađenom funkcijom `update_one`, u kombinaciji sa operacijama modifikacije,[27] ažuriramo delove izabranog dokumenta, ne menjajući ostatak.

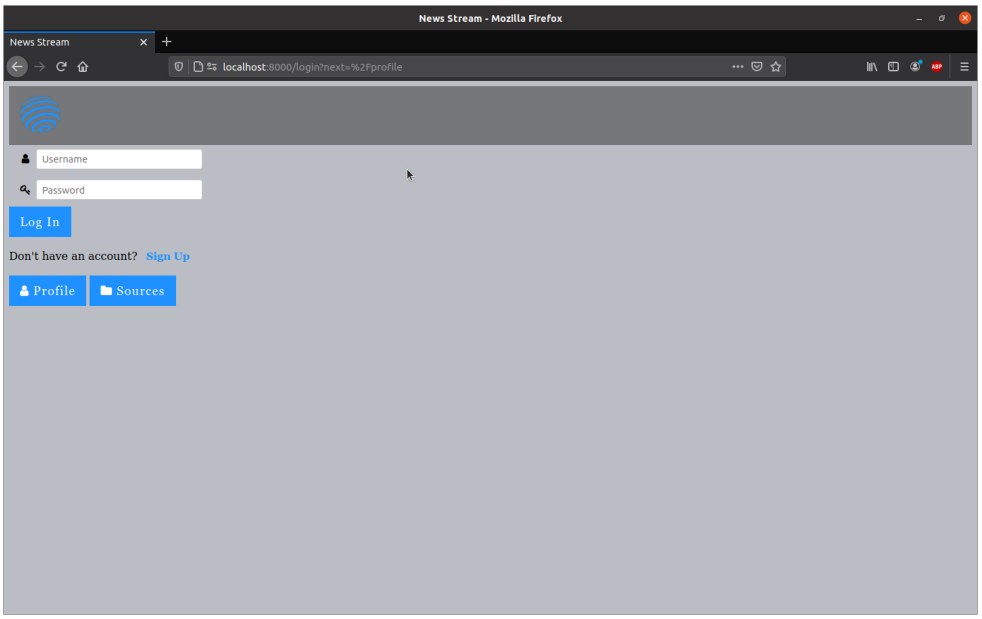
[27] Neke od ponuđenih operacija su `$min`,`$set`,`$push`,`$rename`,`$inc` itd.

```
async def do_find_one(my_collection, value1):
    document = await my_collection.find_one({"name": value1})
    return document


async def do_check_one(my_collection, value1, value2):
    document = await my_collection.find_one({"name": value1, "password": value2})
    return document


async def do_insert(my_collection, value1, value2):
    document = {"name": value1, "password": value2,
                "headlines": [], "timer": 600}
    result = await my_collection.insert_one(document)
    return result


async def do_insert_headlines(my_collection, name, value1, value2, value3, value4):
    document = {"headline": value1, "description": value2,
                "url_headline": value3, "url_img": value4}
    result = await my_collection.update_one({"name": name},
                                            {'$push': {"headlines": document}})
    return result
```

Slika 5.19: Primer korišćenja nekih asinhronih funkcija biblioteke *Motor*

5.6 Upravljanje korisničkim nalozima

Nešto ranije, naveli smo da je korisnik svojom registracijom dobija mogućnost korišćenja još nekih funkcionalnosti aplikacije. Klikom na dugme *Profile* (ili akcijom pokušaja čuvanja vesti) anonimni korisnik se vodi na stranicu za prijavljivanje (Slika 5.20).



Slika 5.20: Prijava postojećih korisnika

Popunjavanjem i slanjem ovog formulara serveru, pokreće se više različitih mehanizama (Slika 5.21).

```
username = self.get_argument("username")
password = self.get_argument("password")

val = await do_check_one(collection, username, password)

if val is not None:
    self.set_secure_cookie("username", username)
    self.redirect("/profile")
else:
    self.write('<h1>Wrong credentials</h1>')
```

Slika 5.21: Sistem proveravanja prijave korisnika

Kako korisnik do sada nije bio upisan, to i njegov pregledač nije imao postavljen kolačić sa vrednošću njegovog korisničkog imena, i u ovom trenutku on se postavlja. Osim toga, proverava se postojanje korisnika sa datim parom korisničkog imena i lozinke u bazi podataka. Po uspešnoj pretrazi, korisnik je odveden na profilnu stranicu, dok se u suprotnom izdaje poruka o grešci. Slična situacija je i sa registrovanjem novog korisnika (Slika 5.22).

```
username = self.get_argument("username")
password = self.get_argument("password")
email = self.get_argument("email")

val = await do_find_one(collection, username)

if val:
    self.write('Username already exists. Please choose other one.')
    return

val1 = await do_insert(collection, username, password)

if val1 is not None:
    self.set_secure_cookie("username", username)
    self.redirect("/profile")
else:
    print("An error occurred while writing into a database.")
```

Slika 5.22: Sistem registrovanja novog korisnika

Nakon prvobitne provere o jedinstvenosti korisničkog imena, u bazu podataka se smeštaju podaci o novoregistrovanom klijentu i postavlja se kolačić.

U svakom slučaju, od sada pa nadalje (zapravo, dok se korisnik ne odjavi sa sitema, kada se i postavljeni kolačić briše(imamo informaciju o tome ko trenutno koristi aplikaciju. U svim klasama rukovaoca, dostupan nam je podatak o trenutnom korisniku kroz promenljivu `self.current_user`, čija se vrednost postavlja nadjačavanjem metoda `get_current_user` u baznom rukovaocu (Slika 5.23).

```
def get_current_user(self):
    uname = self.get_secure_cookie("username")

    if uname:
        return uname.decode("utf-8")
    return None
```

Slika 5.23: Sistem dobijanja podatka o trenutnoj ulogovanosti korisnika

U našem slučaju, ovaj metod čita kolačić korisničkog imena i vraća njegovu vrednost ukoliko je on postavljen, odnosno `None` ukoliko nije. Na taj način obezbedili smo da se bitne informacije (poput korisnikove lozinke) ne čuvaju u kolačiću, već u bazi podataka.

Profilnoj stranici dodeljen je rukovalac iznad čijeg metoda `get` se nalazi dekorator `@tornado.web.authenticated`. Ovo efektivno znači da neverifikovani korisnik[28] nije u mogućnosti da pristupi toj stranici već biva preusmeren na stranicu koju smo naveli u rečniku podešavanja aplikacije. U našem slučaju to je stranica `/login` na kojoj su obezbeđeni mehanizmi prijave i registracije.

Korisniku je obezbeđena odjava klikom na dugme *Log Out* na profilnoj strani (Slika 5.4). Samim klikom pokreće se akcija brisanja kolačića `username` i redirekcija na početnu stranu (Slika 5.24).

```
if self.get_argument("logout", None) is not None:

    self.clear_cookie("username")
    self.redirect("/")
    return
```

Slika 5.24: Sistem odjave korisnika

[28]Korisnik nije verifikovan ako je vrednost promenljive `current_user` postavljena na `None`, tj. ukoliko metod `get_current_user` ima za rezultat `None`.

Poglavlje 6

Zaključak

Posmatrajući razvoj računarstva od njegovih početaka, pa do danas, može se uočiti sledeći obrazac. Iz potrebe dobija se proizvod. No, sa porastom apetita, proizvodi zastarevaju i moraju se unapređivati kako bi ispratili potrebe. Ni razvoj veb tehnologija nije izuzet od ovoga. Počevši od potrebe za automatizovanim deljenjem informacija između naučnika na univerzitetima i institutima širom sveta, kroz kontinualni razvoj (koji još uvek traje) došli smo do toga da ne možemo da zamislimo život bez ovog medijuma komunikacije. Prateći sve više rastuće potrebe, uporedo se razvijaju i tehnologije.

Kroz ovaj rad i propratnu aplikaciju pozabavili smo se pitanjem primene asinhronog pristupa u razvoju veb aplikacija. Sama aplikacija predstavlja informativni servis za pregled trenutnih dešavanja u zemlji, ali i u mnogim zemljama širom sveta. Cilj aplikacije je bio da se prikaže asinhroni pristup u domenu razvoja veb aplikacija korišćenjem radnog okvira *Tornado* programskog jezika *Python*.

Asinhroni pristup u programskom jeziku *Python* pokazao se kao veoma koristan u razvoju veb aplikacija koje moraju da odgovore na veliki broj simultanih zahteva klijenata. Postoji više načina za realizaciju asinhronog pristupa u *Python*-u. Prvi među njima su funkcije povratnog poziva čiji je glavni adut konceptualna jednostavnost. Uvođenjem (ali i daljim razvojem) biblioteke `asyncio` stavljena je tačka na debatu o implementaciji asinhronosti u *Python*-u. Korutine biblioteke `asyncio`, posebno prirodne korutine, postavljaju standard za razvoj asinhronih veb aplikacija u ovom programskom jeziku. Po savlađivanju njihovog načina rada, programer može da piše kod koji izuzetno liči na sekvencijalni kod, dokle god ne zaboravlja da je sistem izvršavanja nedeterministički, kao i da se pri pozivu korutine, kontrola vraća petlji događaja.

Prednosti korišćenja radnog okvira *Tornado*, su višestruke. Sa strane servera, ovaj radni okvir karakteriše implementirana podrška neblokirajućim funkcijama za rad sa ulazno-izlaznim operacijama. Sa klijentske strane, ovaj radni okvir implementira svoj jezik šablona kojima možemo generisati čitave veb stranice, za koje je obezbeđena i podrška dodatnih funkcionalnosti u vidu stilizovanja jezikom *CSS*, ili izvršavanja *JavaScript* koda. Uz to, tu su i neki bezbednosni mehanizmi zaštite podataka poput sigurnih kolačića i zaštite od *XSRF* napada. Od nedostataka koji su uočeni prilikom rada koristeći ovaj radni okvir, izdvojićemo to što sam *Tornado* nema ugrađeni sistem za asinhrono upravljanje bazama podataka, već se oslanja na spoljne biblioteke. Pored toga, sistem autentifikacije korisnika svodi se na čuvanje u kolačiću, što nosi sa sobom određena bezbednosna pitanja.

Ukoliko je potreban server koji ima mogućnost efikasne obrade velikog broja istovremenih konekcija, i fokus je na izdvajanju traženih podataka i njihovom slanju klijentima, *Tornado* predstavlja odličan izbor.

Literatura

[1] *Most Popular Technologies*. 2019. URL: https://insights.stackoverflow.com/survey/2019#most-popular-technologies (visited on 06/22/2020).

[2] *Developer Ecosystem 2019 - Python*. 2019. URL: https://www.jetbrains.com/lp/devecosystem-2019/python/ (visited on 06/22/2020).

[3] *Most Popular Frameworks for Python*. 2019. URL: https://steelkiwi.com/blog/top-10-python-web-frameworks-to-learn/ (visited on 06/22/2020).

[4] Alex Davies. *Async in C# 5.0*. Vol. 1. O'REILLY, 2012, pp. 1–2. URL: https://books.google.rs/books?id=xT45qhFrVnUC&redir_esc=y.

[5] *I/O Bound*. 2003. URL: http://osr507doc.sco.com/en/PERFORM/ident_IO_bound.html (visited on 07/31/2020).

[6] *Monkey patch*. 2005. URL: https://web.archive.org/web/20080604220320/http://plone.org/documentation/glossary/monkeypatch (visited on 07/31/2020).

[7] *epoll*. 2020. URL: https://man7.org/linux/man-pages/man7/epoll.7.html (visited on 07/31/2020).

[8] *kqueue*. 2020. URL: https://people.freebsd.org/~jlemon/papers/kqueue.pdf (visited on 07/31/2020).

[9] *A Brief Timeline of Python*. 2009. URL: http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html (visited on 07/02/2020).

[10] *Personal History - part 1, CWI*. 2009. URL: http://python-history.blogspot.com/2009/01/personal-history-part-1-cwi.html (visited on 07/02/2020).

[11] *PEP 20 – The Zen of Python*. 2004. URL: https://www.python.org/dev/peps/pep-0020/ (visited on 07/02/2020).

[12] *The ABC Programming Language: a short introduction*. 2012. URL: https://homepages.cwi.nl/~steven/abc/ (visited on 07/03/2020).

[13] *10.12. Batteries Included*. 2020. URL: https://docs.python.org/3/tutorial/stdlib.html (visited on 07/03/2020).

[14] *1.10. Reference Counts*. 2020. URL: https://docs.python.org/3/extending/extending.html#reference-counts (visited on 07/03/2020).

[15] *Automated Python 2 to 3 code translation*. 2020. URL: https://docs.python.org/3/library/2to3.html (visited on 07/03/2020).

[16] *Python 2.0*. 2020. URL: https://www.python.org/download/releases/2.0/ (visited on 07/03/2020).

[17] *Python 3.0*. 2020. URL: https://www.python.org/download/releases/3.0/ (visited on 07/03/2020).

[18] Luciano Ramalho. *Fluent Python*. Vol. 1. O'REILLY, 2014, pp. 345–346. URL: https://www.amazon.com/Fluent-Python-Concise-Effective-Programming/dp/1491946008.

[19] *Python 3.0*. 2012. URL: https://docs.python.org/3/tutorial/controlflow.html#defining-functions (visited on 07/07/2020).

[20] Richard Hightower. *Python Programming with the Java Class Libraries*. Vol. 1. Addison-Wesley Professional, 2003, p. 121. URL: https://books.google.rs/books?id=Fw3i6p3yUFkC&hl=sr&source=gbs_navlinks_s.

[21] Luciano Ramalho. *Fluent Python*. Vol. 1. O'REILLY, 2014, p. 751. URL: https://www.amazon.com/Fluent-Python-Concise-Effective-Programming/dp/1491946008.

[22] *Origins of Python's "Functional" Features*. 2009. URL: http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html (visited on 07/13/2020).

[23] *Generators*. 2020. URL: https://wiki.python.org/moin/Generators (visited on 03/07/2020).

[24] *List Comprehensions*. 2020. URL: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions.

[25] *Built-in Exceptions*. 2020. URL: https://docs.python.org/3/library/exceptions.html#concrete-exceptions.

[26] *Built-in Functions*. 2020. URL: https://docs.python.org/3/library/functions.html#built-in-functions.

[27] *Design philosophies*. 2020. URL: https://docs.djangoproject.com/en/3.0/misc/design-philosophies/#don-t-repeat-yourself-dry.

[28] *The C10K problem*. 2020. URL: http://www.kegel.com/c10k.html.

[29] *FriendFeed*. 2020. URL: https://www.crunchbase.com/organization/friendfeed.

[30] *Queues for coroutines*. 2020. URL: https://www.tornadoweb.org/en/stable/queues.html#module-tornado.queues.

[31] *News API - A JSON API for live news and blog articles*. 2020. URL: https://newsapi.org/ (visited on 08/08/2020).

[32] Brian Albers Peter Lubbers Frank Salim. *Pro HTML5 Programming*. Vol. 1. Apress, 2011, p. 352. URL: https://www.amazon.com/Pro-HTML5-Programming-Application-Development/dp/143023864X.

[33] *Torndb Documentation*. 2017. URL: https://torndb.readthedocs.io/en/latest/.

[34] The Tornado Autors. *Tornado Documentation, Release 6.0.4*. Vol. 1. 2020, p. 9. URL: https://www.tornadoweb.org/_/downloads/en/stable/pdf/.

[35] *HMAC: Keyed-Hashing for Message Authentication*. 1997. URL: https://tools.ietf.org/html/rfc2104.

[36] *Welcome to PyMySQL's documentation!* 2016. URL: https://pymysql.readthedocs.io/en/latest/.

[37] *PyMongo 3.11.0 Documentation*. URL: https://pymongo.readthedocs.io/en/stable/#.

[38] *Asynchronous IO Support Rebooted: the "asyncio" Module*. Dec. 12, 2012. URL: https://www.python.org/dev/peps/pep-3156/.

[39] Luciano Ramalho. *Fluent Python - Clear, concise, and effective programming*. Vol. 1. O'REILLY, 2014, p. 481. URL: https://www.amazon.com/Fluent-Python-Concise-Effective-Programming/dp/1491946008.

[40] *PEP 492 – Coroutines with async and await syntax*. Sept. 4, 2015. URL: https://www.python.org/dev/peps/pep-0492/.

[41] *Built-in Exceptions*. Sept. 1, 2020. URL: https://docs.python.org/3/library/exceptions.html#StopIteration (visited on 09/01/2020).