Veb aplikacije u Python-u

Desktop i veb aplikacije

- Aplikacija je bilo koji program koji je napravljen da bi bio pojednostavljen ili obavljen neki zadatak.
- Desktop aplikacija je instalirana na klijentskoj mašini.
- Desktop (klasična) aplikacija se pokreće izvršavanjem određenog izvršnog fajla (exe / jar).
- Veb aplikacija je instalirana na nekom udaljenom server i njoj se pristupa putem internet-a.
- Veb aplikacija se pokreće zahtevom odgovarajućeg URL-a u veb pregledaču,
- koji predstavlja interfejs veb aplikacija.

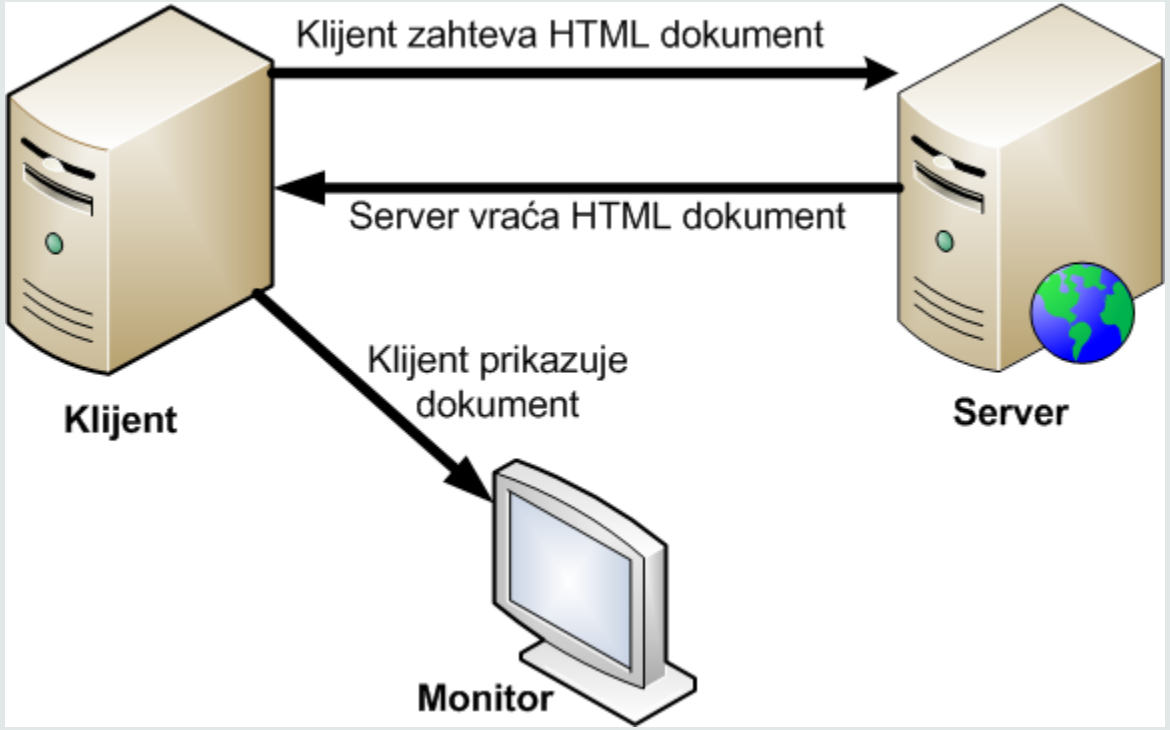
Aplikacije i baze podataka

- Baza podataka se koriste za skladištenje podataka, ako klasična ili veb aplikacija rade sa većom količinom podataka.
- Podaci u bazi podataka su potpuno nezavisni od aplikacije.
- Posebnim naredbama aplikacija čita podatke iz baze podataka i ažurira podatke u bazi podataka (dodaje, briše, menja).

Statički veb sajtovi

- Statički sajtovi se ne menjaju sve dok sam autor nešto ne promeni.
- Omogućavaju slanje informacija ka korisnicima.
- Korisnici nemaju mogućnost interakcije i ne mogu neki zadatak da izvršavaju na programabilan nacin.
- Statičke veb sajtove čine obične statičke HTML stranice (skup HTML fajlova).

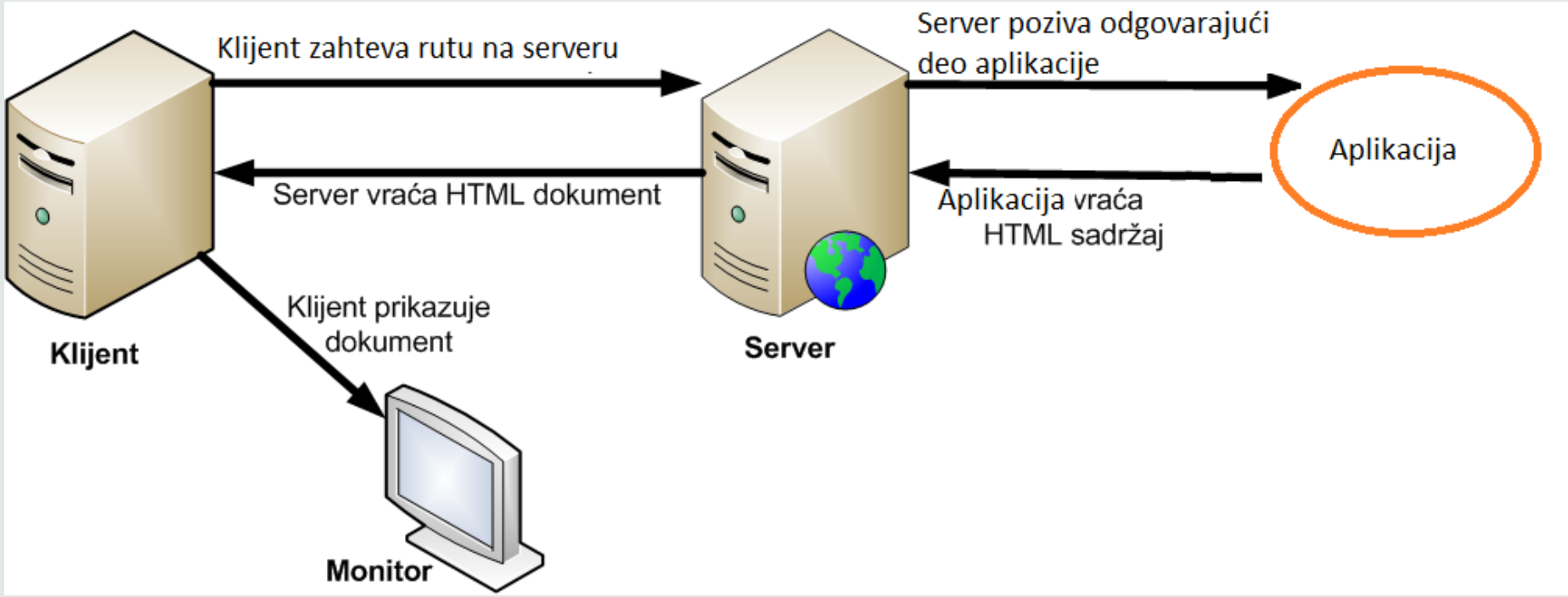
Zahtev za HTML dokumentom



Dinamičke veb aplikacije

- Dinamičke veb aplikacije - interkacija sa korisnikom
- Dinamičke veb aplikacije čine dinamičke stranice (stranice koje su dinamički učitane na osnovu klijentskog ponašanja) koje takođe koriste HTML jezik za komunikaciju sa klijentom
- Slanje zahteva kod klijent-server arhitekture:

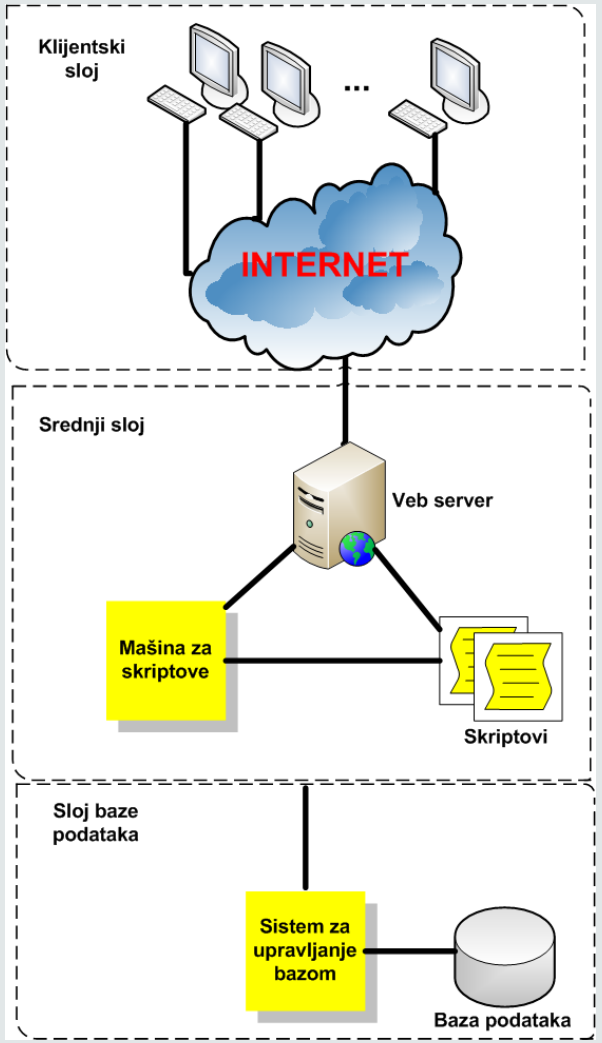
Dinamička veb aplikacija



Arhitektura troslojne veb aplikacije

Većina veb aplikacija koje rade sa bazama podataka poseduje tzv. troslojnu arhitekturu koju čine sledeći slojevi:

- klijentski sloj
- aplikacioni sloj
- sloj baze podataka



Kako su povezani slojevi?

- Sam veb obezbeđuje mrežu i protokole koji povezuju klijentski sloj sa srednjim slojem, a to je dominantno HTTP (HyperText Transfer Protocol).
- Komunikacija između srednjeg sloja i sloja baze podataka se obavlja pomoću protokola zavisnog od tipa programskog jezika i tipa baze podataka.
- Komunikacija preko HTTP-a dominira mrežnim saobraćajem na internetu (oko 80%).
- Za izradu veb aplikacija sa bazama podataka nije potrebno detaljno poznavati HTTP, ali je vrlo važno znati kakve probleme može da izazove HTTP u veb aplikacijama koje rade sa bazama podataka.

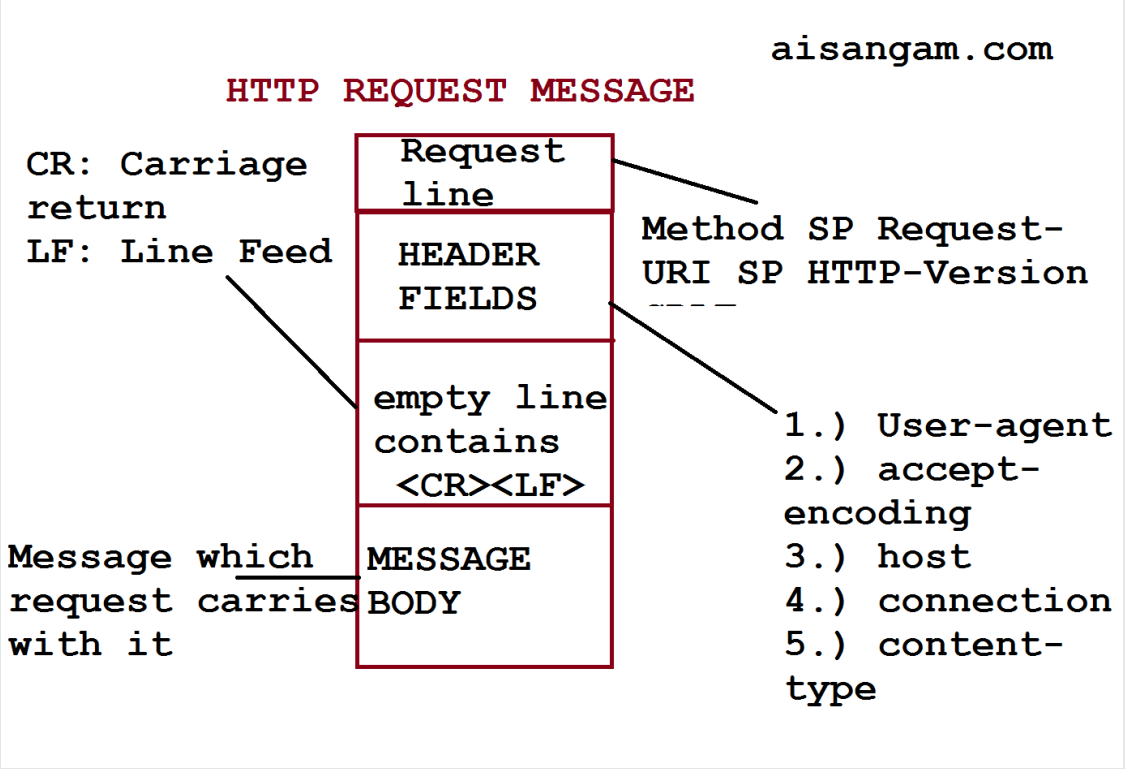
Osnove HTTP-a

- HTTP je standard koji omogućava prosleđivanje i deljenje dokumenata na veb-u.
- Klijent (koji je u većini slučajeva veb pregledač) šalje HTTP serveru zahtev za određenim resursom, a server mu šalje svoj odgovor, u kome je sadržan i sam resurs - najčešće HTML dokument.

Čuvanje stanja

- Klasične aplikacije koje rade sa bazama podataka čuvaju stanje (korisnici se prvo prijavljuju, zatim izvršavaju pojedine transakcije, i na kraju kad obave posao odjavljuju se).
- HTTP ne čuva stanje tj. svaka interakcija između veb pregledača i veb servera je nezavisna od bilo koje druge interakcije.
- Dakle treba na neki način obezbediti čuvanje tekućeg stanja. Za čuvanje tekućeg stanja PHP obezbeđuje kolačiće i sesije.

Struktura HTTP zahteva



Klijentski sloj

Klijentski sloj u modelu troslojne arhitekture je veb pregledač (eng. web browser).

Veb pregledač šalje HTTP zahteve za resursima, obrađuje HTTP odgovore I prikazuje HTML resurse.

Prednosti upotrebe veb pregledača kao tankog klijenta:

- jednostavan razvoj
- nezavisnost od platforme na klijentskoj strani

Najpopularniji veb pregledači: Internet Explorer, Mozilla Firefox, Google Chrome, Opera,...

Veb pregledači (čitači)

Zajedničke odlike pregledača (čitača) veb-a:

- Svi veb pregledači su HTTP klijenti koji šalju zahteve i prikazuju odgovore dobijene od veb servera (obično u nekom grafičkom okruženju)
- Svi čitači tumače sadržaj HTML stranica
- Neki čitači prikazuju slike, reprodukuju filmove i zvuk i vizuelizuju druge tipove objekata
- Mnogi čitači mogu da izvršavaju JavaScript kod ugrađen u HTML stranice (npr. za proveru ispravnosti unetih podataka u HTML formi)
- Nekoliko čitača može da primenjuje kaskadne liste stilova (Cascading Style Sheets, CSS)

Postoje manje razlike u načinima na koji pojedini čitači veb-a prikazuju HTML stranice. Npr. neki ne prikazuju slike ili ne izvršavaju Java Script kod itd.

URL

- URL-ovi (Uniform Resource Locator) koriste se na veb-u kao primarni način za imenovanje i adresiranje resursa.
- Drugim rečima, URL omogućava da se jedinstveno identifikuje adresa nekog dokumenta/fajla na veb-u.
- Sam HTTP standard ne ograničava dužinu URL-a, ali to ipak čine neki stariji veb čitači i posrednički serveri.

Struktura URL

URL se može podeliti na tri osnovna dela:

- identifikator protokola
- identifikator ciljnog računara i servisa na njemu (port)
- putanja (koja može da sadrži još i parametre)

Struktura URL-a:

scheme://host:port/path?parameter=value#anchor

Scheme - Identifikator protokola

Prvi deo URL-a (scheme) služi da se identifikuje aplikacioni protokol.

Najčešće korišćeni protokoli:

- **http** - HyperText Transfer Protocol (protokol za prenos hiperteksta)
- **https** - HyperText Transfer Protocol with SSL (protokol za prenos hiperteksta putem veza zaštićenim SSL protokolom)
- **ftp** - File Transfer Protocol (protokol za prenos fajlova)

Host - identifikator ciljnog računara

- Drugi deo URL-a (host) identifikuje ciljni računar (server).
- host se može zadati preko imena ili kao IP adresa.
- Primeri:

    www.w3.org

    18.29.1.35

Port - identifikator servisa na ciljnom računaru

- Treći deo URL-a (port) specificira TCP port number i služi za identifikaciju servisa na ciljnom računaru.
- Na internetu postoji konvencija da se dobro poznati priključak koristi na serveru koji pruža dobro poznati tip usluge (HTTP server očekuje zahteve na priključku 80, FTP server na priključku 21, itd).
- TCP priključak (port) nije fizički uređaj, već identifikator koji služi TCP softveru i omogućava uspostavljanje više virtuelnih veza sa istom mašinom za različite aplikacije.

Path - identifikator resursa na ciljnom računaru

- Četvrti deo URL-a (path) služi za identifikovanje resursa na ciljnom računaru, i to je putanja (najčešće stvarna putanja do ciljne datoteke na serveru).
- Ako se ime ciljne datoteke izostavi podrazumeva se index.html.

Dodatni parametri u URL-u

- Peti deo URL-a (parameter=value) se koristi za prosleđivanje vrednosti parametara za dodatnu obradu putem URL-a.
- U slučaju prosleđivanja više parova parametar -vrednost, isti se razdvajaju znakom &.
- Primeri GET parametara:

  p1=Beograd

  p1=Beograd&p2=2003

Anchor - identifikator fragmenta

Šesti deo URL-a (*anchor*) predstavlja identifikator fragmenta se koristi za pozicioniranje na određeno mesto u okviru dokumenta.

Ova mogućnost se koristi u slučaju velikih dokumenata.

Mesto "skoka" mora biti registrovano u okviru dokumenta.

Aplikacioni sloj (1/2)

Aplikacioni sloj sadrži aplikacionu logiku koja se uglavnom realizuje preko skriptova.

Skriptovi obavljaju sledeće funkcije:

- Obavljaju određene zadatke karakterististične za svaku aplikaciju pojedinačno (izračunavanja,...).
- Komuniciraju sa DBMS-om na strani veb servera.
- Generišu HTML kod potreban za prezentaciju podataka u korisnikovom veb pregledaču.

Aplikacioni sloj (2/2)

Najveći deo aplikacione logike nalazi se u aplikacionom sloju.

Aplikacioni sloj obavlja većinu zadataka pomoću kojih se objedinjuju ostali slojevi:

- Obrađuje podatke dobijene od korisnika pretvarajući ih u upite za čitanje ili upisivanje u bazu podataka.
- Upravlja se strukturom i sadržajem podataka koji se prikazuju korisniku.
- Omogućava upravljanje stanjem uz upotrebu HTTP protokola.

Sloj baze podataka

Sloj baze podataka se sastoji od sistema za upravljanje bazom podataka (Database Management System - DBMS).

DBMS omogućava:

- Čitanje podataka iz baze
- Ažuriranje podataka u bazi (unos, brisanje i izmena)

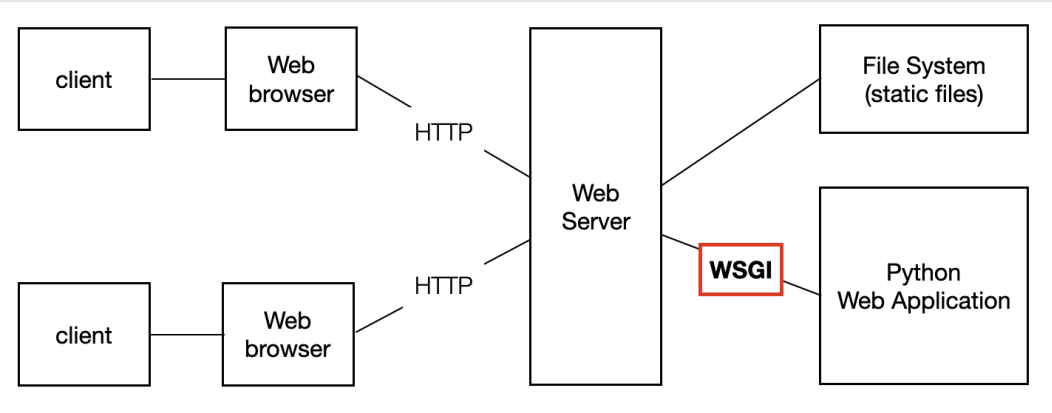
CRUD - Create, Read, Update, Delete

Organizacija prema objektima (entitetima) i odnosima koje postoje u sistemu na koji se baza podataka odnosi.

Krajnji cilj integrisanosti je minimalna redudansa (višestruko pojavljivanje) podataka.

Veb server u Python-u

- Za implementaciju servera koristiće se WSGI (Web Server Gateway Interface) server.
- Može se posmatrati kao apstrakcija, koja uprošćava primanje HTTP zahteva sa serverske strane i omogućuje da ne moramo da implementiramo serversku logiku od nule.
- Biblioteka koja će se koristiti za ovo je **wsgiref**.



WSGI server i wsgiref

Prati WSGI standard specificiran u PEP 0333

Biblioteka **wsgiref** obrađuje zahteve sa klijentske strane i prosleđuje ih serverskoj aplikaciji koja služi za poslovnu logiku (backend)

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.handle_request()
```

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.handle_request()
```

Adresa gde se podiže server.
Localhost je sinonim za lokalni server,
tj. računar gde se aplikacija pokreće trenutno.
Alternativna localhost imenu je IP adresa 127.0.0.1.

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.handle_request()
```

Port na kome će raditi veb aplikacija.

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.handle_request()
```

Aplikacija kojoj će WSGI server prosleđivati HTTP zahteve.

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.handle_request()
```

Server se pokreće i prestaje da radi nakon tačno jednog primljenog HTTP zahteva.

Kreiranje WSGI servera

```
if __name__ == '__main__':
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s. serve_forever()
```

Server radi beskonačno, odnosno dok ne bude prekinut iz nekog razloga eksplicitno.

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

Svaka WSGI aplikacija mora da ima tačno dva parametra:
environ – Environment Dictionary, gde stoje sve bitne informacije o HTTP zahtevu
start_response – Callback funkcija koja se obavezno zove da bi se ispravno izgenerisao HTTP odgovor

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

Poziv start_response funkcije, koja ima prima dva argumenta: status HTTP odgovora (status), i zaglavlja HTTP odgovora (headers).

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

Status označava informaciju servera klijentu o “vrsti” HTTP odgovora, odnosno kako klijent treba da ga protumači.

HTTP statusi

Propisana je konvencija značenja statusnih kodova po RFC 9110, gde su odgovori grupisani u pet klasa:

- Informational responses (statusi 100-199)
- Successful responses (statusi 200-299)
- Redirection messages (statusi 300-399)
- Client error responses (statusi 400-499)
- Server error responses (statusi 500-599)

HTTP konkretne vrednosti statusa

Od interesa će biti samo nekoliko statusnih kodova odgovora, a to su (detaljnije na linku):

- 200 OK – Zahtev je uspešno procesiran
- 404 Not Found – Server ne može da pronađe zahtevani resurs (Zahtevana je nepostojeća slika npr.)
- 500 Internal Server Error – Server je naišao na grešku koju ne zna kako da obradi (najčešće je došlo do greške u kodu, tj. implementaciji)

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

HTTP odgovor ima različite vrste zaglavlja koji se definiše u vidu niza Tupple-ova sa tačno dve vrednosti: prva je samo zaglavlje koje se definiše, a drugo vrednost.

HTTP zaglavlja odgovora

Postoji ogroman broj HTTP zaglavlja (i za odgovore i zahteve)

Cela lista se može videti na sledećem linku

Za sada će od interesa biti dva:

- Content-Length – definiše dužina resursa u odgovoru u broju bajtova
- Content-Type – definiše vrstu resursa koja se vraća odgovorom (u slučaju html stranice, text/html je vrednost)

Izgled proste veb aplikacije

```
def application(environ, start_response):
    response = b"Hello World"
    status = "200 OK"
    headers = [("Content-Type", "text/html")]
    start_response(status, headers)
    return [response]
```

Kao povratnu vrednost funkcije u vidu niza definišemo sam HTTP odgovor (njegovo telo) u vidu niza bajtova. U ovom primeru se vraća HTML stranica sa Hello World sadržajem

Ali ovo nije ni približno funkcionalnoj veb aplikaciji...

Šta sve nedostaje?

- Neka vrsta logike koja će sugerisati u zavisnosti od prosleđene putanje koji HTML sadržaj vraćamo sa servera
- Povratna informacija koja vraća korisniku informaciju u slučaju nepostojeće stranice

Izgled malo naprednije, ali opet proste veb aplikacije

```
def application(environ, start_response):


    with open("index.html", "r") as f:
        response = f.read().encode()
    status = "200 OK"



    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return [response]
```

Radi lakše implementacije, učitan je sadržaj same stranice iz fajla i prebačen u niz bajtova

Izgled malo naprednije, ali opet proste veb aplikacije

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    if path == "/":
        with open("index.html", "r") as f:
            response = f.read().encode()
        status = "200 OK"



    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return [response]
```

Pod ključem PATH_INFO stoji informacija o putanji (ruti) koju klijent zahteva na serveru.

Izgled malo naprednije, ali opet proste veb aplikacije

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    if path == "/":
        with open("index.html", "r") as f:
            response = f.read().encode()
        status = "200 OK"



    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return [response]
```

Možemo onda proverom da vidimo da li je korena putanja u pitanju, i samo u tom slučaju da vratimo sadržaj index.html stranice.

Izgled malo naprednije, ali opet proste veb aplikacije

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    if path == "/":
        with open("index.html", "r") as f:
            response = f.read().encode()
        status = "200 OK"
    else:
        response = b"<h1>Not Found</h1><p>Entered path not found</p>"
        status = "404 Not Found"
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return [response]
```

U slučaju bilo koje druge zahtevane rute (/aa npr), vratiće se stranica sa informacijom o greški. I status će biti 404, jer ne postoji zaista zahtevani resurs na toj putanji.

Još uvek nema obrade sadržaja unetog od strane klijenta...

Nedostaje obrada informacija koje klijent može da unese i koje se šalju serveru: obrada za unos teksta, padajući meniji itd.

To jest, polja u okviru tzv. formi.

Klikom na dugme unutar forme se generiše HTTP zahtev

Primer forme u index.html stranici

```
<form action="/results" method="post">
   <select name="proizvodi">
      <option value="p1">Proizvod1</option>
      <option value="p2">Proizvod2</option>
   </select>
   <br>
   <input type="number" name="broj">
   <input type="submit" name="Dodaj proizvod">
</form>
```

Primer forme u index.html stranici

```
<form action="/results" method="post">
  <select name="proizvodi">
    <option value="p1">Proizvod1</option>
    <option value="p2">Proizvod2</option>
  </select>
  <br>
  <input type="number" name="broj">
  <input type="submit" name="Dodaj proizvod">
</form>
```

Primer forme u index.html stranici

```
<form action="/results" method="post">
   <select name="proizvodi">
      <option value="p1">Proizvod1</option>
      <option value="p2">Proizvod2</option>
   </select>
   <br>
   <input type="number" name="broj">
   <input type="submit" name="Dodaj proizvod">
</form>
```

Ovaj tip dugmeta generiše HTTP zahtev koji prenosi unete podatke u formi serveru.

Primer forme u index.html stranici

```
<form action="/results" method="post">
   <select name="proizvodi">
      <option value="p1">Proizvod1</option>
      <option value="p2">Proizvod2</option>
   </select>
   <br>
   <input type="number" name="broj">
   <input type="submit" name="Dodaj proizvod">
</form>
```

Ruta na serveru koja se gađa (u ovom slučaju http://localhost:8021/results)

Primer forme u index.html stranici

```
<form action="/results" method="post">
   <select name="proizvodi">
      <option value="p1">Proizvod1</option>
      <option value="p2">Proizvod2</option>
   </select>
   <br>
   <input type="number" name="broj">
   <input type="submit" name="Dodaj proizvod">
</form>
```

Način (metod) kako se prosleđuju podaci u HTTP zahtevu. Postoje dve osnovne vrste:
GET i POST.

HTTP GET i POST zahtevi

U zavisnosti od toga da li je za HTTP zahtev specificirano da li je GET ili POST zavisi kako će se prosleđivati parametri serveru (u ovom slučaju oni uneti u formi).

- Ako je GET – prosleđuju se kroz URL serveru (npr. http://localhost:8021/results?proizvodi=p1&broj=1)
- Ako je POST- prosleđuju se kroz telo HTTP zahteva (npr. telo zahteva će imati sadržaj: proizvodi=p1&broj=1)

Ostali HTTP metodi

Osim ove dve vrste, HTTP još nekoliko različitih vrsta metoda:

- CONNECT, DELETE, HEAD, OPTIONS, PATCH, PUT, TRACE

Uglavnom rade slično kao POST metod i imaju samo semantičku razliku

GET vs POST

Kada korstiti koji?

- GET koristiti za situacije gde se šalje manje podataka serveru, jer je dućina URL-a limitirana, za razliku od tela HTTP zahteva
- POST koristiti kad se šalje veliki broj podataka (uglavnom forme) i kada se šalju poverljive informacije (lozinke npr.)

Šta želimo da se desi nakon stiska dugmeta?

Kada korisnik preda podatke serveru, želimo da pređemo na drugu stranicu (results.html), gde će mu se prikazati uneti podaci.

Kako će onda izgledati server?

Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
```

Ključ pod kojim se na serveru dohvata informacija o metodu HTTP zahteva.

Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":

        request_body_raw =
        environ['wsgi.input'].read(                    )
```

Ukoliko je POST, čitamo telo HTTP zahteva koje stoji pod wsgi.input ključem.

Izgled servera

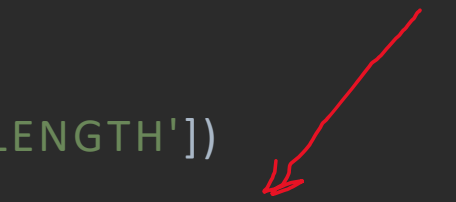
```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw =
        environ['wsgi.input'].read(request_body_size)
```

Međutim, to se prosleđuje kao niz bajtova, pa je zato neophodno znati koliko bajtova je potrebno pročitati...

Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw =
        environ['wsgi.input'].read(request_body_size).decode('utf-8')
```

...i konvertovati u string na kraju iz niza bajtova.



Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw =
        environ['wsgi.input'].read(request_body_size).decode('utf-8')
        request_body = dict(re.findall(r'([^=.]+)=([^=.]+)(?:&|$)',
                                       request_body_raw))
```

Na kraju želimo telo HTTP zahteva da konvertujemo u dictionary, jer sadrži vrednosti parametara prosleđenih kroz formu putem regex-a...

Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw =
        environ['wsgi.input'].read(request_body_size).decode('utf-8')
        request_body = dict(re.findall(r'([^=.]+)=([^=.]+)(?:&|$)',
                                       request_body_raw))
```

...tako što se telo sa sadržajem **proizvodi=p1&broj=1** prevodi u dictionary: {'proizvodi': 'p1', 'broj': '1'}

Izgled servera

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw =
        environ['wsgi.input'].read(request_body_size).decode('utf-8')
        request_body = dict(re.findall(r'([^=.]+)=([^=.]+)(?:&|$)',
                                       request_body_raw))
```

Prikazani regex radi tako što radi izdvajanje stringova pre i posle ‘=‘, razdvojenih ili sa ‘&’ ili krajem stringa. Ovo će biti svi parovi polja u formi i njihovih vrednosti.

Nastavak servera

```
if path == "/":
    with open("index.html", "r") as f:
        response = f.read().encode()
    status = "200 OK"
else:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

Ostatak servera je neophodno izmeniti u skladu sa dodatim izmenama.

1. Mapiranje rute i stranice koja se vraća korisniku
2. Dinamičko generisanje HTML sadržaja na osnovu parametara

Rešavanje problema #1

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
routes = {
    '/': "index.html",
    '/results': "results.html"
}
```

Rešavanje problema #1

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

Ukoliko ne postoji ključ u routes-u, dešava se izuzetak.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

Sadržaj ne treba učitavati predefinisano kao ovde iz fajla, već nekako “uglaviti” prosleđene parametre.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
response = response.format(**data)
```

Sadržaj ne treba učitavati predefinisano kao ovde iz fajla, već nekako “uglaviti” prosleđene parametre.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
response = response.format(**data)
```

Argument data predstavlja sadržaj u formi dictionary koji treba da se prosledi stringu koji se formatira (u ovom slučaju HTML stranici).

Izgled results.html stranice

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Results</title>
</head>
<body>
 <h1>Second Page! {proizvod} </h1>
 <p> Number: {num} </p>
</body>
</html>
```

Stvari koje stoje u { } su zamišljene da budu zamenjene odgovarajućim vrednostima prosleđenog dictionary objekta putem format metode.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
data = dict()
data["proizvod"] = request_body['proizvodi']
data["num"] = request_body['broj']
response = response.format(**data)
```

Na osnovu izgleda results.html, potrebno je proslediti dictionary sa ključevima proizvod i num.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>"
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
if path == "/results":
    data = dict()
    data["proizvod"] = request_body['proizvodi']
    data["num"] = request_body['broj']
    response = response.format(**data)
```

Ovo treba raditi samo u slučaju da je ruta /results.

Rešavanje problema #2

```
try:
    with open(routes[path], "r") as f:
        response = f.read().encode()
    status = "200 OK"
except KeyError:
    response = b"<h1>Not Found</h1>
                 <p>Entered path not found</p>'
    status = "404 Not Found"
headers = [
    ("Content-Type", "text/html"),
    ("Content-Legth", str(len(response)))
]
start_response(status, headers)
return [response]
```

```
response = f.read()
if path == "/results":
    data = dict()
    data["proizvod"] = request_body['proizvodi']
    data["num"] = request_body['broj']
    response = response.format(**data)
response = response.encode()
```

Nedostaci ovakvog pristupa

Na osnovu ovako male aplikacije, već se primećuje efekat otežavanja izmena sa skaliranjem veličine aplikacije.

Kod je težak za održavanje, loše strukturiran i težak za izmenu.

Kako razrešiti ovakav problem? Modularizacijom koda!

Koji bi to bili moduli onda?

Uvođenje modularizacije aplikacije

U uvodu su se izvdvojile tri celine aplikacije:

- Klijentski sloj
- Aplikacioni sloj
- Sloj sa podacima

Možemo da pratimo ove celine i tako podelimo našu aplikaciju. I to upravo radi arhitekturalni projektni uzorak...

MVC – Model View Controller

Projektni uzorak koji prati veliki broj radnih okvira za izradu veb aplikacija.

Sastoji se iz tri celine:

- Model – sloj rada sa podacima (bazom podataka)
- View – sloj za prikazivanje podataka (HTML sadržaja)
- Controller – sloj biznis logike, koji povezuje Model i View

Šta želimo da naša aplikacija radi?

1. Na naslovnoj strani postoji lista proizvoda od kojih se može da se izabere jedan, zajedno sa količinom koju želimo
2. Prelaskom na narednu stranicu se prikazuje šta smo uneli (sam proizvod i njegov opis, tj. narudžbina)
3. Sa naslovne strane je moguće da se pređe na drugu gde se se prikazuje forma za unos novog proizvoda, za koji je moguće da se odabere na naslovnoj strani iz padajućeg menija, nakon unosa.

Kako prebaciti našu aplikaciju u MVC?

1. Deo logike servera koji prima i parsira HTTP zahtev za sada ostaje relativno nepromenjen.
2. Na osnovu zahtevane rute, umesto što se pravi mapiranje na samu HTML stranicu koja treba da se učita, sada se radi mapiranje ka funkcijama (kontrolerima) koji rade odgovarajuću obradu
3. Svaku komunikaciju sa podacima skladištimo u modele (za sada samo in-memory podaci)

Korak #1 prebacivanja aplikacije u MVC

```
def application(environ, start_response):
    path = environ["PATH_INFO"]
    method = environ["REQUEST_METHOD"]
    request = dict()
    request['body'] = {}
    if method == "POST":
        request_body_size = int(environ['CONTENT_LENGTH'])
        request_body_raw = str(environ['wsgi.input'].read(request_body_size).decode('utf-8'))
        request_body = dict(re.findall(r'([^=.]+)=([^=.]+)(?:&|$)', request_body_raw))
        for key, value in request_body.items():
            request_body[key] = value.replace("+", " ")
        request['body'] = request_body
    response = render_response_page(path, request, start_response)
    return [response]
```

Dodata funkcija koja služi za mapiranje rute i generisanje HTTP odgovora.

Izgled funkcije render_response_page

```
def render_response_page(path, request, start_response):
    try:
        page, data = routes[path](request)
        with open(page, "r") as f:
            response = f.read()
            if data is not None:
                response = response.format(**data)
            response = response.encode()
        status = "200 OK"
    except KeyError:
        response = b"<h1>Not Found</h1><p>Entered path not found</p>"
        status = "404 Not Found"
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return response
```

Izgled funkcije render_response_page

```
def render_response_page(path, request, start_response):
    try:
        page, data = routes[path](request)
        with open(page, "r") as f:
            response = f.read()
            if data is not None:
                response = response.format(**data)
            response = response.encode()
        status = "200 OK"
    except KeyError:
        response = b"<h1>Not Found</h1><p>Entered path not found</p>"
        status = "404 Not Found"
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return response
```

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Ove funkcije su definisane u fajlu za kontrolere.
Za svaku je zamišljeno da prima kao jedan argument objekat koji predstavlja uprošćen HTTP zahtev...

Izgled funkcije render_response_page

```
def render_response_page(path, request, start_response):
    try:
        page, data = routes[path](request)
        with open(page, "r") as f:
            response = f.read()
            if data is not None:
                response = response.format(**data)
            response = response.encode()
        status = "200 OK"
    except KeyError:
        response = b"<h1>Not Found</h1><p>Entered path not found</p>"
        status = "404 Not Found"
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response)))
    ]
    start_response(status, headers)
    return response
```

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

...a kao povratnu vrednost da vraća HTML stranicu koja treba da se učita i podatke koji treba da joj se proslede.

Korak #2 prebacivanja aplikacije u MVC

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Korak #2 prebacivanja aplikacije u MVC

```
def index(request):
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi}



routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Kontroler za učitavanje index.html stranice (početne stranice). Proizvodi se učitavaju iz lokalne “baze” i prosleđuju među povratnim vrednostima, jer treba da se dinamički ugrade u HTML sadržaj.

Korak #2 prebacivanja aplikacije u MVC

```
def results(request):
    proizvod = models.dohvati_proizvod(request["body"]["proizvodi"])
    return "results.html", {"proizvod": proizvod['ime'] + ": " + proizvod["opis"]}
```

Kontroler za učitavanje results.html stranice. Poziva se nakon predaje forme serveru (/results je ruta specificirana formom). Informacije o proizvodu se učitavaju iz lokalne “baze” i prosleđuju među povratnim vrednostima, jer treba da se dinamički ugrade u HTML sadržaj.

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Korak #2 prebacivanja aplikacije u MVC

```
def view_add_proizvod(request):
    return "add_proizvod.html", {}
```

Kontroler za učitavanje add_proizvod.html stranice, gde se nalazi forma za unos proizvoda. Poziva se pritiskom linka na naslovnoj stranici.

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Korak #2 prebacivanja aplikacije u MVC

```
def add_proizvod(request):
    models.dodaj_proizvod(request["body"]["ime"], request["body"]["opis"])
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi}


routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod
}
```

Kontroler za unos proizvoda, nakon čega se prelazi na index.html stranicu. Poziva se nakon predaje forme serveru. Prosleđene informacije o proizvodu se u lokalnu “bazu”.

Korak #3 prebacivanja aplikacije u MVC

```
proizvodi = {
   "p1": {
      "ime": "Proizvod 1",
      "opis": "Opis proizvoda 1"
   },
   "p2": {
      "ime": "Proizvod 2",
      "opis": "Opis proizvoda 2"
   }
}
```

Lokalna in-memory “baza” je implementirana putem dictionary-ja, ali će sama promena na stvarnu bazu podataka biti jednostavna.

Korak #3 prebacivanja aplikacije u MVC

```
def dohvati_sve_proizvode():
    return proizvodi.items()


def dohvati_proizvod(id_proizvod):
    return proizvodi[id_proizvod]

def dodaj_proizvod(ime, opis):
    id = "p" + str(len(proizvodi) + 1)
    proizvodi[id] = {"ime": ime, "opis": opis}
```

Funkcije za rad sa lokalnom bazom, koje koriste kontroleri.
U slučaju prave baze podataka će na ovim mestima biti pisani upiti.

HTTP ne čuva stanje

Veb pregledač i veb server međusobno komuniciraju pomoću HTTP protokola koji ne čuva stanje između dve razmene podataka.

Svaki HTTP zahtev koji veb pregledač šalje veb serveru nezavisan je od svih drugih zahteva.

Ovakva komunikacija veb pregledača i veb servera pogodna je za aplikacije koje korisnicima omogućavaju da pretražuju ili pregledaju grupe povezanih dokumenata koristeći hiperlinkove.

Potreba za čuvanjem stanja

U aplikacijama u kojima je potrebna složenija intervencija korisnika, mora se obezbediti čuvanje stanja pri prelasku sa jedne stranice aplikacije na drugu.

Primer: Veb aplikacija prodavnice

Korisnik dodaje stavke u korpu dok pretražuje ili pregleda neki katalog.

Stanje korpe za kupovinu (stavke koje su sadržane u njoj) mora se negde čuvati.

Kada korisnik zahteva stranicu za prikaz sadržaja korpe, mora se prikazati koje se stavke u njoj nalaze.

Čuvanje stanja bez upotrebe sesije

U svakom HTTP zahtevu, među podacima koji se šalju metodom GET ili POST, prosleđivati i podatke koji predstavljaju trenutno stanje aplikacije.

Nepotrebno povećanje saobraćaja na veb-u.

Ako se podaci koji opisuju stanje prenose HTTP GET metodom (kao deo URL-a), korisnik može ručno da izmeni vrednosti koje se šalju zahtevom, a često nastaju i dugačke i nepregledne URL adrese.

Čuvanje stanja - kolačići i sesije

Stanje aplikacije (vrednosti pojedinih promenljivih) mora se negde skladištiti između dva HTTP zahteva.

Promenljive koje opisuju stanje mogu se čuvati na dva mesta:

1. u klijentskom veb pregledaču
2. na veb serveru

Konretno za našu aplikaciju...

Potrebno je dodati sledeću funkcionalnost:

- Stanje porudžbine koju je korisnik napravio (na naslvonoj stranici), treba da ostane zapamćena, osim ako je korisnik ne zameni drugom porudžbinom
- Dodatno se uvode linkovi na drugim stranicama za pristup stranici za porudžbinu (results.html).
- Ukoliko još uvek ništa nije naručeno, na stranici za porudžbinu se ispisuje odgovarajuća poruka o tome.

Na osnovu zahteva koji je tražen...

Jasno je da je potrebno nekako čuvati stanje o tome da se zapamtila porudžbina.

Koristićemo kolačiće za te potrebe.

Šta treba da se izmeni u našoj aplikaciji da bi omogućili kolačiće?

Kako ubaciti kolačiće u aplikaciju?

1. Ukoliko postoje, kolačići koji čuvaju informaciju o porudžbini treba da se proslede serveru, i server u skladu sa time treba da izgeneriše stranicu za prikaz porudžbine (results.html).
2. Nakon slanja zahteva serveru u formi za porudžbinu (index.html), potrebno je da se dodatno zapamti informacija o porudžbini kroz kolačić.

Korak #1 u korišćenju kolačića

```
def application(environ, start_response):
    ...
    if 'HTTP_COOKIE' in environ:
        request['cookies'] = environ["HTTP_COOKIE"]
    else:
        request['cookies'] = dict()
    response = render_response_page(path, request, start_response)
    return [response]
```

Ukoliko postoje kolačići zapamćeni sa klijentske strane, biće prosleđeni serveru, a u environ se pamte pod ključem HTTP_COOKIE.

Korak #1 u korišćenju kolačića

```
def view_cart(request):
    cookie = cookies.SimpleCookie(request['cookies'])
    try:
        proizvod = models.dohvati_proizvod(cookie['proizvodi'].value)
        return "results.html", {"proizvod": proizvod['ime'] + ": " + proizvod["opis"], "broj":
                                cookie['broj'].value}, []
    except KeyError:
        return "results.html", {"proizvod": "Vasa korpa je prazna!", "broj": ""}, []
```

Stranica za prikaz korpe.

Korak #1 u korišćenju kolačića

```
def view_cart(request):
    cookie = cookies.SimpleCookie(request['cookies'])
    try:
        proizvod = models.dohvati_proizvod(cookie['proizvodi'].value)
        return "results.html", {"proizvod": proizvod['ime'] + ": " + proizvod["opis"], "broj":
                                cookie['broj'].value}, []
    except KeyError:
        return "results.html", {"proizvod": "Vasa korpa je prazna!", "broj": ""}, []
```

Koristiće se biblioteka SimpleCookie
za učitavanje prosleđenih kolačića.

Korak #1 u korišćenju kolačića

```
def view_cart(request):
    cookie = cookies.SimpleCookie(request['cookies'])
    try:
        proizvod = models.dohvati_proizvod(cookie['proizvodi'].value)
        return "results.html", {"proizvod": proizvod['ime'] + ": " + proizvod["opis"], "broj":
                                cookie['broj'].value}, []
    except KeyError:
        return "results.html", {"proizvod": "Vasa korpa je prazna!", "broj": ""}, []
```

Value atribut služi za dohvatanje vrednosti kolačića u stringu pod odgovarajućim ključem.

Korak #2 u korišćenju kolačića

```
def render_response_page(path, request, start_response):
   cookie_output = []
   try:
      page, data, cookie_output = routes[path](request)

      ...
      status = "200 OK"
   except KeyError:
      ...
   headers = [
      ("Content-Type", "text/html"),
      ("Content-Legth", str(len(response))),
   ]
   for output in cookie_output:
      headers.append(
         ("Set-Cookie", output)
      )
   start_response(status, headers)
   return response
```

Korak #2 u korišćenju kolačića

```
def render_response_page(path, request, start_response):
    cookie_output = []
    try:
        page, data, cookie_output = routes[path](request)

        ...
        status = "200 OK"
    except KeyError:
        ...
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response))),
    ]
    for output in cookie_output:
        headers.append(
            ("Set-Cookie", output)
        )
    start_response(status, headers)
    return response
```

Uvodi se još jedna povratna vrednost svakog kontrolera, niz kolačića koji treba da se postavi sa klijentske strane.

Korak #2 u korišćenju kolačića

```
def render_response_page(path, request, start_response):
    cookie_output = []
    try:
        page, data, cookie_output = routes[path](request)

        ...
        status = "200 OK"
    except KeyError:
        ...
    headers = [
        ("Content-Type", "text/html"),
        ("Content-Legth", str(len(response))),
    ]
    for output in cookie_output:
        headers.append(
            ("Set-Cookie", output)
        )
    start_response(status, headers)
    return response
```

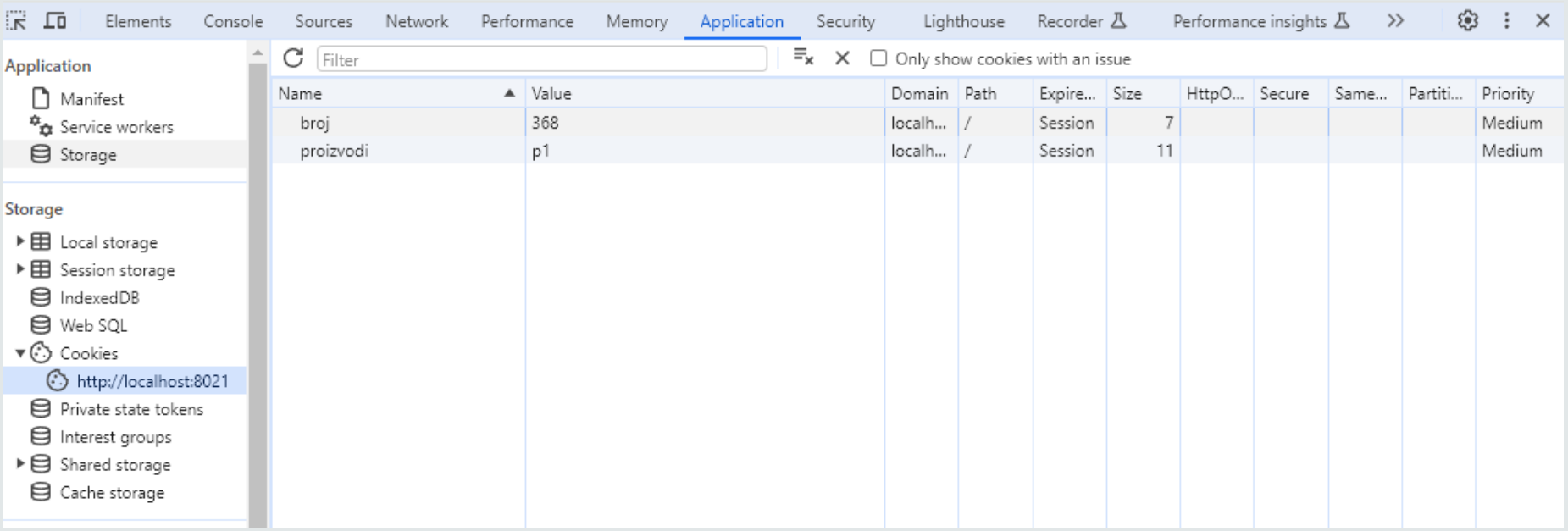
Putem Set-Cookie zaglavlja u HTTP odgovoru, klijentu (pretraživaču) se specificira da treba da zapamti informaciju prosleđenu kao vrednost. To će se odraditi za svaki element niza koji je vraćen od strane kontrolera.

Korak #2 u korišćenju kolačića

```
def results(request):
    cookie = cookies.SimpleCookie(request['cookies'])
    cookie['proizvodi'] = request["body"]["proizvodi"]
    cookie['broj'] = request["body"]["broj"]
    proizvod = models.dohvati_proizvod(cookie['proizvodi'].value)
    return "results.html",
   {"proizvod": proizvod['ime'] + ": " + proizvod["opis"], "broj": cookie['broj'].value},
   [cookie['proizvodi'].output(header=''), cookie['broj'].output(header='')]
```

Informacije o kolačićima se prosleđuju kroz niz, korišćenjem SimpleCookie metode output, koja vraća string izlaz, za odgovarajući kolačić.

Efekat postavljanja kolačića sa klijentske strane



Osnovne informacije kolačića

- **name**: naziv kolačića; obavezan parametar
- **value**: vrednost kolačića koja će se čuvati na računaru klijenta; ne čuvati poverljive informacije
- **expire**: vreme u timestamp formatu kada kolačić prestaje da važi
- **path**: putanja na serveru na kojoj će kolačić biti dostupan
- **domain**: domen u kome je kolačić dostupan
- **secure**: označava da kolačić treba da bude poslat samo putem sigurne HTTPS konekcije

Ograničenja za kolačiće (1/2)

Kolačići se mogu koristiti u jednostavnim aplikacijama u kojima nije neophodno da se složeni podaci čuvaju između dva zahteva serveru.

Broj i veličina kolačića su ograničeni:

veb pregledač može da čuva samo poslednjih 20 kolačića koji su mu bili poslati iz određenog domena, a veličina svakog kolačića je ograničena na 4KB (ovo se vremenom menja i zavisi od verzije pregledača).

Ograničenja za kolačiće (2/2)

Pitanje privatnosti korisnika i zaštite aplikacije u kojima se koriste kolačići.

Neki korisnici isključuju podršku za rad sa kolačićima.

Upravljanje sesijama na vebu

Podaci o tekućem stanju aplikacije čuvaju se na veb serveru tj. u srednjem sloju aplikacije.

Rešava se problem čuvanja promenljivih stanja koje zauzimaju više prostora i/ili većeg broja promenljivih stanja.

Rešava se problem zaštite podataka sadržanih u promenljivama stanja od nenamernih ili namernih izmena koje bi korisnik mogao načiniti.

Sesija

Sesija (engl. session) je jedan od načina označavanja i upravljanja skupom podataka o stanju, pomoću sesijskih promenljivih datog korisnika

Kada korisnik pošalje HTTP zahtev, srednji sloj aplikacije mora da obradi tekući zahtev vodeći računa o kontekstu (stanju) sesije.

Početak sesije

Kada korisnik započne sesiju, klijentskom pregledaču šalje se identifikator sesije, najčešće u obliku kolačića, čija se vrednost ugrađuje u sve naredne zahteve koje veb pregledač upućuje serveru.

Pomoću identifikatora sesije server identifikuje odgovarajuću sesiju pre nego što nastavi dalju obradu prispelog zahteva.

Identifikator sesije

Kod kolačića se lokalno (u klijentskom veb pregledaču) skladište vrednosti svih promenljivih koje su neophodne za održavanje stanja i one se ugrađuju u svaki HTTP zahtev.

Ako se koriste sesije veb pregledač čuva i ugrađuje u svaki HTTP zahtev samo identifikator sesije, na osnovu koga se jednoznačno identifikuju sesijske promenljive njegove sesije.

Problem zamrznutih sesija (1/3)

U srednjem sloju se čuvaju zasebni podaci za svaku sesiju.

Pitanje: Koliko dugo?

Kada se sve odvija kako bi trebalo korisnik se sam odjavljuje iz aplikacije (recimo klikom na dugme "Kraj rada"), a skript koji se onda pokreće završava sesiju.

Međutim, u ovo se ne smete pouzdati. Korisnici se često ne odjavljuju iz aplikacije na adekvatan način.

Problem zamrznutih sesija (2/3)

Ako se korisnik ne odjavi adekvatno iz aplikacije, njegova sesija se neće završiti tj. njegove sesijske promenljive će se i dalje čuvati na serveru.

Server nikada ne može da bude siguran da li se na drugoj strani veze još uvek nalazi korisnik (svaki HTTP zahtev je nezavisan od drugih zahteva).

Zato server treba redovno da čisti stare, nezavršene (zamrznute) sesije u kojima se tokom određenog vremena nije ništa događalo.

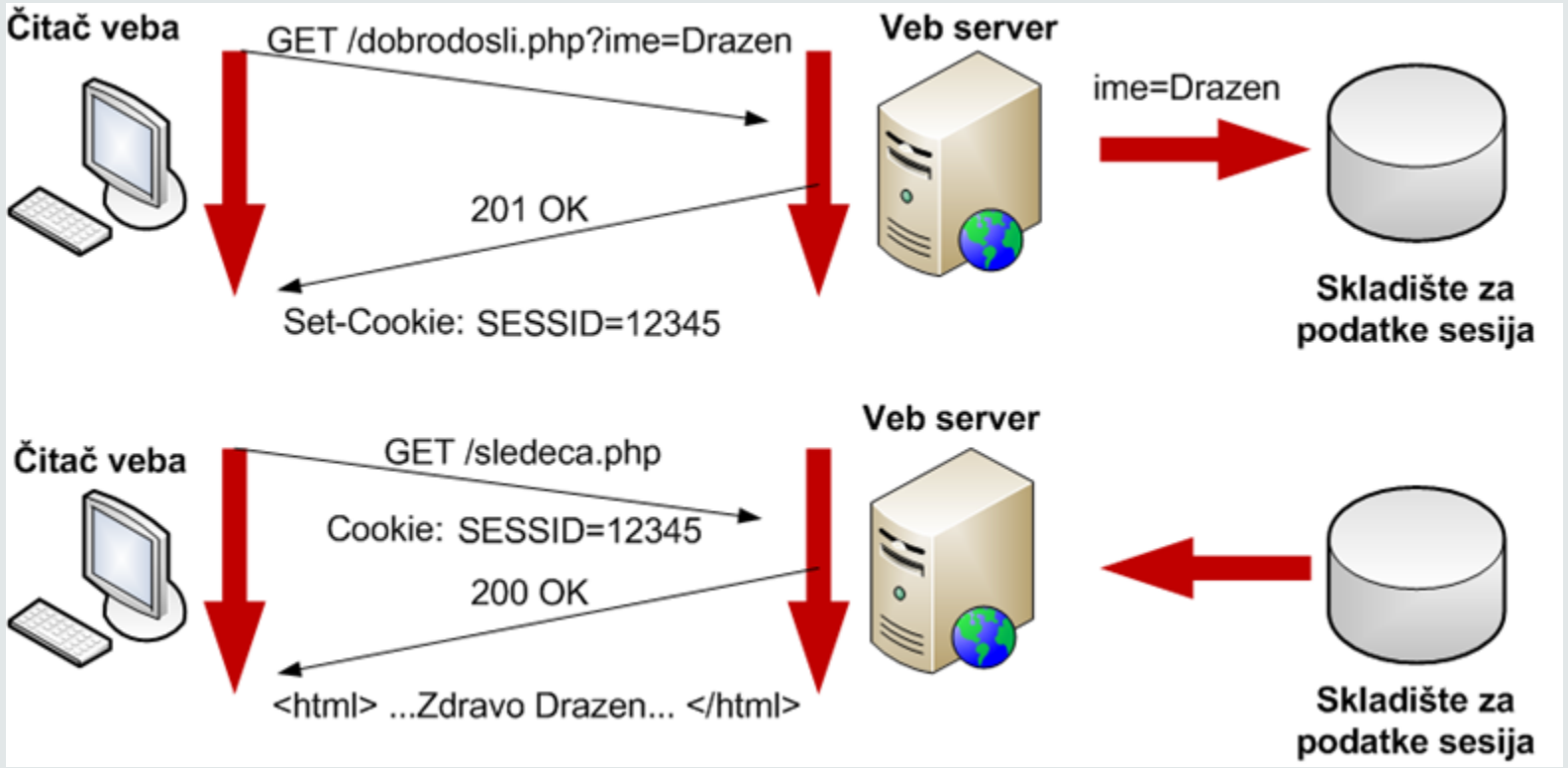
Problem zamrznutih sesija (3/3)

Problemi koje izazivaju zamrznute sesije:

- troše resurse servera
- bezbedonosni rizik

Dužina intervala čekanja, pre nego što se neka zamrznuta sesija očisti, nije univerzalni parametar i zavisi od potreba aplikacije.

Mehanizam sesije



Gde čuvati sesijske promenljive?

Moguće je upisivanje vrednosti sesijskih promenljivih u datoteke na disku, što je pogodno za većinu aplikacija.

Rešenje koje omogućava veći stepen prilagodljivosti, zasnovano je na upotrebu MySQL-ove baze podataka za čuvanje podataka o sesijama (samo za jako zahtevne aplikacije gde je broj i veličina sesijskih promenljivih izražena).

Potrebna nam je funkcionalnost u aplikaciji...

- Potrebna nam je funkcionalnost prijave u sistem.
- Ukoliko je korisnik prijavljen, prikazuje mu se korisničko ime na naslovnoj stranici.
- Za ovo su nam potrebne sesije!

Kako dodati sesije u aplikaciju?

1. Pamćenje sesije u sistemu, kada se korisnik prijavi.
2. Prikaz informacija korisnika, ukoliko postoji sesija.
3. Pamćenje sesije trajno (u datoteku), ukoliko se server restartuje.

Korak #1 u korišćenju sesija

```
def login_check(request):
    korisnik = models.login(request['body']['korime'], request['body']['lozinka'])
    if korisnik != {}:
        cookie = cookies.SimpleCookie(request['cookies'])
        session_id = sessionutils.generate_session_id()
        sessionutils.sessions[session_id] = dict()
        sessionutils.sessions[session_id]['korisnik'] =
          models.korisnici[request['body']['korime']]
        cookie['session_id'] = session_id
        sessionutils.save_to_file(session_id)
        session_cookie_output = cookie['session_id'].output(header='')
        proizvodi = ""
        for k, v in models.dohvati_sve_proizvode():
            proizvodi += f"<option value='{k}'>{v['ime']}</option>"
        return "index.html", {"proizvodi": proizvodi, "korime": request['body']['korime']}, [session_cookie_output]
    return "login.html", {"poruka": "<p style='color:red'>Neispravni kredencijali!</p>"}, []
```

Ovaj kontroler se poziva u situaciji kada korisnik proba da se prijavi u sistem.

Korak #1 u korišćenju sesija

```
def login_check(request):
    korisnik = models.login(request['body']['korime'], request['body']['lozinka'])
    if korisnik != {}:
        cookie = cookies.SimpleCookie(request['cookies'])
        session_id = sessionutils.generate_session_id()
        sessionutils.sessions[session_id] = dict()
        sessionutils.sessions[session_id]['korisnik'] =
          models.korisnici[request['body']['korime']]
        cookie['session_id'] = session_id
        sessionutils.save_to_file(session_id)
        session_cookie_output = cookie['session_id'].output(header='')
        proizvodi = ""
        for k, v in models.dohvati_sve_proizvode():
            proizvodi += f"<option value='{k}'>{v['ime']}</option>"
        return "index.html", {"proizvodi": proizvodi, "korime": request['body']['korime']}, [session_cookie_output]
    return "login.html", {"poruka": "<p style='color:red'>Neispravni kredencijali!</p>"}, []
```

U slučaju da postoji korisnik, generiše se unikatni sesijski identifikator, koji se vraća kao kolačić k

Korak #1 u korišćenju sesija

```
def login_check(request):
    korisnik = models.login(request['body']['korime'], request['body']['lozinka'])
    if korisnik != {}:
        cookie = cookies.SimpleCookie(request['cookies'])
        session_id = sessionutils.generate_session_id()
        sessionutils.sessions[session_id] = dict()
        sessionutils.sessions[session_id]['korisnik'] =
          models.korisnici[request['body']['korime']]
        cookie['session_id'] = session_id
        sessionutils.save_to_file(session_id)
        session_cookie_output = cookie['session_id'].output(header='')
        proizvodi = ""
        for k, v in models.dohvati_sve_proizvode():
            proizvodi += f"<option value='{k}'>{v['ime']}</option>"
        return "index.html", {"proizvodi": proizvodi, "korime": request['body']['korime']}, [session_cookie_output]
    return "login.html", {"poruka": "<p style='color:red'>Neispravni kredencijali!</p>"}, []
```

Da bi se olakšao i bolje dekomponovao proces, uvodi se pomoćni fajl za rad sa sesijama: sessionutils.

Izgled sessionutils fajla

```
sessions = {}
```

Dictionary koji služi za
čuvanje sesija.

Izgled sessionutils fajla

```
def generate_session_id():
    import uuid
    return str(uuid.uuid4())
```

Funkcija za generisanje sesijskog identifikatora.

Izgled sessionutils fajla

```
def save_to_file(session_id):
    if os.path.exists('./tmp/sess_' + session_id):
        f = open('./tmp/sess_' + session_id, 'w')
        json.dump(sessions[session_id], f)
    else:
        f = open('./tmp/sess_' + session_id, 'x')
        json.dump(sessions[session_id], f)
```

Funkcija za čuvanje sadržaja sesije u fajl (sa prosleđenim sesijskim identifikatorom). Ovde se ova funkcija poziva samo nakon prijave, pa se uzima da se ta sesija neće menjati u međuvremenu, osim ako se korisnik ne prijavi ponovo.

Izgled sessionutils fajla

```
def save_to_file(session_id):
    if os.path.exists('./tmp/sess_' + session_id):
        f = open('./tmp/sess_' + session_id, 'w')
        json.dump(sessions[session_id], f)
    else:
        f = open('./tmp/sess_' + session_id, 'x')
        json.dump(sessions[session_id], f)
```

U slučaju da ne postonji već fajl za odgovarajuću sesiju kreira se novi, u suprotnom se otvara već postojeći. Naziv fajla predstavlja:
sess_ i sesijski identifikator.

Izgled sessionutils fajla

```
def read_from_files():
    for file in os.listdir("./tmp"):
        session_id = file.split("_")[1]
        f = open("./tmp/" + file)
        sessions[session_id] = json.load(f)
```

Funkcija za učitavanje zapamćenih sesija iz fajlova. Sve sesije stoje u /tmp folderu i za svaku se identifikator određuje na osnovu naziva fajla.

Korak #2 u korišćenju sesija

```
def index(request):
    korime = ""
    cookie = cookies.SimpleCookie(request['cookies'])
    if "session_id" in cookie:
        session_id = cookie['session_id'].value
        if "korisnik" in sessionutils.sessions[session_id]:
            korime = sessionutils.sessions[session_id]['korisnik']['korime']
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi, "korime": korime}, []
```

Rad sa čitanjem iz sesije radi kroz par koraka:

Korak #2 u korišćenju sesija

```
def index(request):
    korime = ""
    cookie = cookies.SimpleCookie(request['cookies'])
    if "session_id" in cookie:
        session_id = cookie['session_id'].value
        if "korisnik" in sessionutils.sessions[session_id]:
            korime = sessionutils.sessions[session_id]['korisnik']['korime']
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi, "korime": korime}, []
```

Provera da li postoji prosleđen kolačić sa sesijskim identifikatorom (session_id).

Korak #2 u korišćenju sesija

```
def index(request):
    korime = ""
    cookie = cookies.SimpleCookie(request['cookies'])
    if "session_id" in cookie:
        session_id = cookie['session_id'].value
        if "korisnik" in sessionutils.sessions[session_id]:
            korime = sessionutils.sessions[session_id]['korisnik']['korime']
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi, "korime": korime}, []
```

Dohvatanje sesije (koja je već pročitana iz fajla po potrebi) na osnovu identifikatora.

Korak #2 u korišćenju sesija

```
def index(request):
    korime = ""
    cookie = cookies.SimpleCookie(request['cookies'])
    if "session_id" in cookie:
        session_id = cookie['session_id'].value
        if "korisnik" in sessionutils.sessions[session_id]:
            korime = sessionutils.sessions[session_id]['korisnik']['korime']
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return "index.html", {"proizvodi": proizvodi, "korime": korime}, []
```

Ukoliko postoji, dohvatanje informacije pod odgovarajućim ključem.

Korak #3 u korišćenju sesija

```
if __name__ == '__main__':
    sessionutils.read_from_files()
    w_s = wsgiref.simple_server.make_server(
        host="localhost",
        port=8021,
        app=application
    )
    w_s.serve_forever()
```

Prilikom pokretanja servera, sesije je neophodno ažurirati na osnovu informacija iz fajlova (korišćenjem već prikazane funkcije iz sessionutils fajla).

Rad sa bazom

Do sada su primeri koristili lokalnu bazu podataka.

Međutim, želimo da podaci budu perzistentni.

Ovo je moguće raditi upisom u fajlove, što je u redu za aplikacije manjeg obima, ili da ih čuvamo u bazi podataka.

Šta je u načelu baza podataka

Ništa drugo do još jednog servera, koji služi za skladištenje podataka.

Isto ima informacije kao što su host i port.

Da bi smo mi kao aplikacija mogli da komuniciramo sa bazom, ove informacije su nam neophodne (kao što moramo da znamo URL sajta da bi smo mu pristupili iz pretraživača).

Takođe nam trebaju informacije o kredencijalima za pristup bazi.

Pristup bazi podataka

I naravno... sama baza podataka kojoj želimo da pristupimo na serveru.

Postoje različite implementacije sistema sa radom sa bazama, ovde će biti prikazan rad sa MySQL bazom.

Svaka od njih ima svoj način kako se radi konekcija, a i samim tim biblioteke koje rade to za nas.

Ovde će se koristiti biblioteka mysql-connector-python

Konekcija sa bazom

```
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="1234",
  database="proizvodi"
)
```

Konekcija sa bazom se radi putem ove metode i vraća se objekat koji predstavlja instancu konekcije sa bazom, koji će se koristiti za kreiranje upita.

Konekcija sa bazom

```
def dohvati_sve_proizvode():
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    myresult = mycursor.fetchall()
    proizvodi = dict((x, {"ime": y, "opis": z}) for x, y, z in myresult)
    return proizvodi.items()
```

Potrebno je kreirati kursor nad instancom baze koji služi za kreiranje upita.

Konekcija sa bazom

```
def dohvati_sve_proizvode():
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    myresult = mycursor.fetchall()
    proizvodi = dict((x, {"ime": y, "opis": z}) for x, y, z in myresult)
    return proizvodi.items()
```

Koristi se execute metoda kojoj se prosleđuje sam upit u formi stringa.

Konekcija sa bazom

```
def dohvati_sve_proizvode():
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    myresult = mycursor.fetchall()
    proizvodi = dict((x, {"ime": y, "opis": z}) for x, y, z in myresult)
    return proizvodi.items()
```

U slučaju da se očekuje više redova kao povratna vrednost upita, koristi se fetchall metoda, alternativno se koristi fetchone.

Konekcija sa bazom

```
def dohvati_sve_proizvode():
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    myresult = mycursor.fetchall()
    proizvodi = dict((x, {"ime": y, "opis": z}) for x, y, z in myresult)
    return proizvodi.items()
```

Redovi baze se vraćaju u formi niza tupple-ova, gde svaki element jednog predstavlja odgovarajuću kolonu u bazi podataka.

Konekcija sa bazom

```
def dohvati_sve_proizvode():
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    myresult = mycursor.fetchall()
    proizvodi = dict((x, {"ime": y, "opis": z}) for x, y, z in myresult)
    return proizvodi.items()
```

Radi lakšeg baratanja sa podacima, i uniformnosti sa prethodnim rešenjem, moguće je odraditi konverziju niza tupple-ova u dictionary.

Konekcija sa bazom

```
def dodaj_proizvod(ime, opis):
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    id = str(len(mycursor.fetchall()) + 1)
    mycursor.execute("INSERT INTO proizvodi (idproizvodi, ime, opis) VALUES (%s, %s, %s)", (id, ime, opis))
    mydb.commit()
```

Upit za unos podataka u bazu slično radi.

Konekcija sa bazom

```
def dodaj_proizvod(ime, opis):
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    id = str(len(mycursor.fetchall()) + 1)
    mycursor.execute("INSERT INTO proizvodi (idproizvodi, ime, opis) VALUES (%s, %s, %s)", (id, ime, opis))
    mydb.commit()
```

Moguće je execute metodi da se prosledi tupple kao drugi argument koji se ubacuje u upit (na mesta gde je %s).

Konekcija sa bazom

```
def dodaj_proizvod(ime, opis):
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM proizvodi")
    id = str(len(mycursor.fetchall()) + 1)
    mycursor.execute("INSERT INTO proizvodi (idproizvodi, ime, opis) VALUES (%s, %s, %s)", (id, ime, opis))
    mydb.commit()
```

Da bi se odradio unos,
poziva se commit umesto
neke verzije fetch metode.

AJAX

AJAX = Asynchronous JavaScript and XML

AJAX je tehnika za kreiranje brzih i dinamičkih veb stranica.

AJAX dozvoljava veb stranicama da se menjaju asinhrono izmenom male količine podataka.

Komunikacija sa serverom odvija u pozadini i na taj način je moguće menjati delove stranice, a ne celu stranicu

Klasične veb stranice (koje ne koriste AJAX) moraju menjati celu stranicu ako se bilo koji deo stranice promeni (reload)

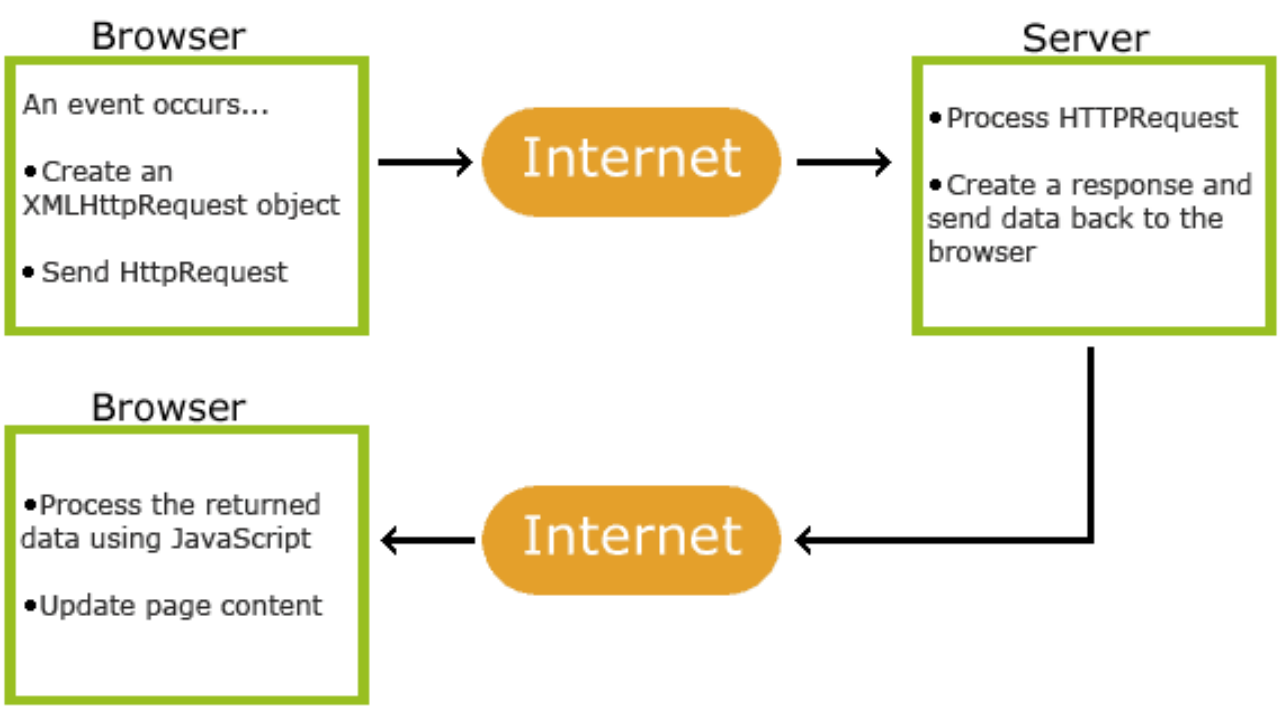
Primeri aplikacija koje koriste AJAX: Google Maps, Gmail, YouTube, Facebook

AJAX

Ranije se kreirao putem klase XMLHttpRequest, sada je zamenjen sa fetch metodom u JavaScript-u, ali je koncept rada još uvek isti.

Sa klijentske strane se kreira HTTP zahtev koji se pošalje serveru i server u skladu sa tipom zahteva vrati odgovarajući sadržaj (najčešće ažurirane podatke).

AJAX mehanizam rada



Kako bismo mogli da iskoristimo ovo?

Potrebno je omogućiti da se prilikom unosa novog proizvoda automatski ažurira lista proizvoda iz koje korisnik može da odabere neki za porudžbinu, umesto nakon osvežavanja stranice, da mu se tek pokažu.

Za ovo možemo AJAX da iskoristimo!

Kako iskoristiti AJAX u aplikaciji

Sa klijentske strane je potrebno da se povremeno šalje asinhroni zahtev serveru o tome koji svi proizvodi postoje.

Samim tim, kada se doda novi proizvod, povući će se informacija sa novododatim proizvodom.

Dodavanje AJAX-a

```
setInterval(ajax_request, 3000);
```

JavaScript nudi funkciju setInterval, koja omogućuje da se periodično zove neka funkcija (kod nas će to biti ajax_request, koji će slati asinhroni zahtev). Ovde će se to raditi na 3 sekunde.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

Kao argument se navodi ruta na serveru koja se gađa. Opciono, drugi argument može da bude sam HTTP zahtev.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

S obzirom na to da se sa fetch radi asinhrono slanje HTTP zahteva, želimo da odradimo ažuriranje tek kada dobijemo podatke nazad.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

Pošto ne znamo tačan trenutak kada će se to desiti, možemo da uradimo await fetch, što će blokirati izvršavanje dok se ne vrate podaci. Međutim, ovime gubimo veliki deo benefita asinhronog izvršavanja, pogotovo ako potraje izvršavanje funkcije.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

Zato se poziva .then od fetch funkcije, kojoj se prosleđuje callback funkcija koja opisuje šta treba da se desi kada se dobije odgovor. Callback funkcija ima jedan parametar, a to je HTTP odgovor od servera.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

Pristup sadržaju tela u formi teksta HTTP odgovora traje, i radi se sa metodom text, pa se opet prosleđuje callback funkcija, u kojoj se opisuje ponašanje kada se dobije odgovor funkcije text.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

U njoj se samo dohvata element koji ima ime proizvodi i upisuje dohvaćeni sadržaj sa servera.

Dodavanje AJAX-a

```
function ajax_request() {{
    fetch("/get-proizvodi")
    .then(response =>
            response.text().then(
                response => document.getElementsByName('proizvodi')[0].innerHTML = response
            )
    )
    .catch(err => console.log(err))
}}
```

U njoj se samo dohvata element koji ima ime proizvodi i upisuje dohvaćeni sadržaj sa servera.

```
<select name="proizvodi">
    {proizvodi}
</select>
```

Kako izgleda ponašanje sa serverske strane?

```
def dohvatiProizvode(request):
    proizvodi = ""
    for k, v in models.dohvati_sve_proizvode():
        proizvodi += f"<option value='{k}'>{v['ime']}</option>"
    return proizvodi
```

Sama ruta je mapirana na ovu funkciju, koja vraća HTML sadržaj sa svim proizvodima na već viđen način.

Kako radi mapiranje?

S obzirom na to da naš sever u ovim situacijama ne vraća HTML stranicu, nama ne treba poziv render_response_page funkcije.

Samim tim, moramo da izdvojimo mapiranje ruta koje su vezane za zahteve generisane putem AJAX-a.

Zato uvodimo još jedan dictionary ruta, apis, koji služi za mapiranje zahteva generisanih putem AJAX-a.

Kako izgleda ponašanje sa serverske strane?

```
routes = {
    '/': index,
    '/results': results,
    '/add-proizvod': add_proizvod,
    '/view-add-proizvod': view_add_proizvod,
    '/view-cart' : view_cart,
    '/login': login,
    '/login-check' : login_check
}

apis = {
    '/get-proizvodi': dohvatiProizvode
}
```

Kako prepoznajemo da je AJAX HTTP zahtev u pitanju?

Jedno od zaglavlja koje se šalje kod HTTP zahteva je Sec-Fetch-Dest (više o zaglavlju na linku).

Služi pošiljaocu HTTP zahteva da naznači kako će iskoristiti vraćen resurs.

U slučaju generisanja HTTP zahteva putem fetch (tj. generisanja AJAX HTTP zahtva), vrednost je postavljena na empty, pa ovo može da se iskoristi.

Kako prepoznajemo da je AJAX HTTP zahtev u pitanju?

```
if environ['HTTP_SEC_FETCH_DEST'] == "empty":
    headers = [
        ("Content-Type", "text/plain"),
    ]
    status = "200 OK"
    start_response(status, headers)
    response = apis[path](request).encode()
else:
    response = render_response_page(path, request, start_response)
```

U ovoj situaciji (AJAX) se generiše običan tekstualni sadržaj, pa je zaglavlje za odgovor postavljeno na text/plain.

Kraj