Tarek Ziadé

# Python

## Razvoj mikroservisa

Izgradnja, testiranje, raspoređivanje i skaliranje mikroservisa u Pythonu

kompjuter biblioteka

Packt›

Tarek Ziadé

# Python

## Razvoj mikroservisa

kompjuter biblioteka

Packt›

**Izdavač:**

kompjuter biblioteka

Obalskih radnika 15, Beograd

**Tel: 011/2520272**

**e-mail:** kombib@gmail.com

**internet:** www.kombib.rs

**Urednik:** Mihailo J. Šolajić

**Za izdavača, direktor:**
Mihailo J. Šolajić

**Autor:** Tarek Ziadé

**Prevod:** Slavica Prudkov

**Lektura:** Miloš Jevtović

**Slog:** Zvonko Aleksić

**Znak Kompjuter biblioteke:**
Miloš Milosavljević

**Štampa:** Apollo Plus D.O.O.
Beograd

**Tiraž:** 500

**Godina izdanja:** 2017.

**Broj knjige:** 497

**Izdanje:** Prvo

**ISBN:** 978-86-7310-520-8

Python Microservices Development
by Tarek Ziadé

ISBN 978-1-78588-111-4

# O AUTORU

**Tarek Ziadé** je Python programer, a živi u jednom selu blizu Dijona u Francuskoj. Radi u servisnom timu u kompaniji „Mozilla“. Osnovao je French Python korisničku grupu pod nazivom „Afpy“ i napisao je nekoliko knjiga o Python jeziku na francuskom i engleskom jeziku. Kada ne hakuje na svom računaru ili nije sa porodicom, provodi vreme između svoje dve strasti - trčanja i sviranja trube.

Možete da posetite njegov lični blog (Fetchez le Python) i da ga pratite na Twitteru (@tarek_ziade). Takođe možete da pogledate jednu od njegovih knjiga na Amazonu „Expert Python Programming“, čiji je izdavač „Packt“.

*Zahvaljujem se za pomoć „Packt“ timu i petočlanoj grupi hakera koju su činili Stéfane Fermigier, William Kahn-Greene, Chris Kolosiwsky, Julien Vehent i Ryan Kelly.*

*Takođe se zahvaljujem Amini, Milou, Suki i Freyi za njihovu ljubav i strpljenje.*

*Nadam se da ćete uživati u ovoj knjizi koliko sam i ja uživao dok sam je pisao.*

# O RECENZENTU

**William Kahn-Greene** koristi Python i programira aplikacije na Vebu od kasnih 1990-ih godina. On radi u grupi crash-stats za kanale za obradu pada sistema u kompaniji „Mozilla“ i održava različite Python biblioteke, uključujući bleach. Kada čeka CI za testiranje promena koda, on pravi skulpture od drveta, neguje svoje biljke paradajza i kuva za četvoro ljudi.

# UVOD

Ako pokušamo da rasporedimo veb aplikacije u Cloud, potrebno je da kod vrši interakciju sa mnogim nezavisnim servisima. Upotrebom arhitektura mikroservisa možemo da izgradimo aplikacije koje će nam omogućiti da upravljamo ovim interakcijama. Međutim, to ima svoje izazove, jer svaki skup ima svoje kompleksnosti i dobro funkcionisanje interakcija nije jednostavno. U ovom jednostavnom vodiču opisane su tehnike koje će vam pomoći da prevaziđete te izazove. Naučićete kako da najbolje projektujete, pišete, testirate i rasporedite mikroservise. Primeri iz stvarnog sveta će pomoći Python programerima da kreiraju sopstvene Python mikroservise korišćenjem najefikasnijih metoda. Do kraja ove knjige steći ćete veštine potrebne za kreiranje aplikacija koje su izgrađene kao male standardne jedinice upotrebom dokazane najbolje prakse i za izbegavanje uobičajenih zamki. Osim toga, ovo je koristan vodič za različite zajednice Python programera koji se prebacuju sa monolitnog projektovanja na novu paradigmu razvoja zasnovanu na mikroservisima.

## ŠTA OBUHVATA OVA KNJIGA?

U Poglavlju 1, *Razumevanje mikroservisa*, definisano je šta su mikroservisi i koje su njihove uloge u modernim veb aplikacijama. Takođe je predstavljen Python i objašnjeno je zašto je on odličan za izgradnju mikroservisa.

U Poglavlju 2, *Otkrivanje Flaska*, predstavljeni su Flask i prolazi kroz njegove osnovne funkcije. Prikazan je radni okvir sa primerom veb aplikacije koja će biti osnova za izgradnju mikroservisa.

Poglavlje 3, *Kodiranje, testiranje i dokumentovanje – virtuelni ciklus*, sadrži opis pristupa razvoja vođenog testiranjem koda i kontinualnom integracijom i način upotrebe u praksi za izgradnju paketnih Flask aplikacija.

U Poglavlju 4, *Projektovanje Runnerlyja*, vodimo vas kroz funkcije aplikacije, objašnjavamo kako ona može da bude izgrađena kao monolitna aplikacija, zatim je rastavljamo na mikroservise i objašnjavamo kako oni vrše interakciju sa podacima. Takođe predstavljamo Open API 2.0 specifikaciju (bivši Swagger), koja može da se upotrebi za opisivanje HTTP API-ja.

U Poglavlju 5, *Interakcija sa drugim servisima*, objašnjeno je kako servis vrši interakciju sa pozadinskim servisima, kako se rešavaju mrežna razdvajanja i drugi problemi interakcije i kako se testira servis u izolaciji.

U Poglavlju 6, *Obezbeđivanje servisa*, objašnjeno je kako se obezbeđuju mikroservisi i kako se rešava provera identiteta korisnika, provera identiteta servis-za-servis i upravljanje korisnicima. Čitaocima su takođe predstavljene prevare i zloupotrebe da bi mogli da ih ublaže.

U Poglavlju 7, *Praćenje servisa*, objašnjeno je kako se dodaju evidentiranje i statistika u kod i kako možemo da se uverimo da imamo jasno globalno razumevanje onoga što se dešava u aplikaciji da bismo mogli da pratimo probleme i razumemo upotrebu servisa.

U Poglavlju 8, *Spajanje*, opisano je kako se projektuje i gradi JavaScript aplikacija koja usklađuje i koristi mikroservise u interfejsu krajnjeg korisnika.

Poglavlje 9, *Pakovanje i pokretanje Runnerlyja*, sadrži opis kako možemo da upakujemo, izgradimo i pokrenemo celu Forrest aplikaciju. Kao programeri, važno je da možemo da pokrenemo sve delove koji čine aplikaciju u jedan dev box.

U Poglavlju 10, *Kontejnerski servisi*, objašnjeno je šta je virtuelizacija, kako se koristi Docker i kako se postavljaju servisi u Docker.

U Poglavlju 11, *Raspoređivanje u AWS*, predstavljeni su postojeći provajderi cloud servisa, a zatim i svet AWS-a, i prikazano je kako se instanciraju serveri i upotrebljavaju glavni AWS servisi koji su korisni za pokretanje aplikacija zasnovanih na mikroservisima. Takođe je predstavljen CoreOS, Linux distribucija koja je specifično kreirana za raspoređivanje Docker kontejnera u cloud.

Poglavlje 12, *Šta dalje?*, sadrži nagovešaje kako mikroservisi mogu da budu izgrađeni, nezavisno od konkretnih cloud provajdera i tehnologija virtuelizacije, za izbegavanje zamki „postavljanja svih jaja u istu korpu“.

# ŠTA VAM JE POTREBNO ZA OVU KNJIGU?

Da biste izvršili komande i aplikacije u ovoj knjizi, potrebno je da na sistemu imate instalirane Python 3.x, Virtualenv 1.x i Docker CE. Detaljne instrukcije su date u poglavljima gde je to potrebno.

# ZA KOGA JE OVA KNJIGA?

Ako ste programer koji ima osnovno znanje o Pythonu, komandnim linijama i principu aplikacija zasnovanih na HTTP-u i ako želite da naučite kako da gradite, testirate i skalirate Python 3 mikroservise i da upravljate njima, onda je ova knjiga za vas. Nije vam neophodno prethodno iskustvo u pisanju mikroservisa u Pythonu.

# KONVENCIJE

U ovoj knjizi pronaći ćete više različitih stilova za tekst koje sam upotrebio za različite vrste informacija. Evo nekih primera ovih stilova i objašnjenja njihovog značenja.

Reči koda u tekstu, nazivi tabela baze podataka, nazivi direktorijuma, nazivi fajlova, ekstenzije fajla, nazivi putanja, kratki URL-ovi, korisnički unos i tweetovi su prikazani na sledeći način: „Jedini znak da koristimo async je ključna reč async, koja označava funkciju za obradu kao korutinu.“

Blok koda je postavljen na sledeći način:

```
import time

def application(environ, start_response):
    headers = [(‚Content-type‘, ‚application/json‘)]
    start_response(‚200 OK‘, headers)
return bytes(json.dumps({‚time‘: time.time()}), ‚utf8‘)
```

Kada želimo da privučemo vašu pažnju na određeni deo bloka koda, relevantne linije ili stavke će biti ispisane masnim slovima:

```
from greenlet
import greenlet
def test1(x, y):
    z = gr2.switch(x+y)
    print(z)
```

Svaki unos komandne linije ili ispis napisan je ovako:

```
docker-compose up
```

**Novi termini** i **važne reči** su napisani masnim slovima.


Upozorenja ili važne napomene će biti prikazani u ovakvom okviru.


Saveti i trikovi prikazani su ovako.

# POVRATNE INFORMACIJE OD ČITALACA

Povratne informacije od naših čitalaca su uvek dobrodošle. Obavestite nas šta mislite o ovoj knjizi – šta vam se dopalo ili šta vam se možda nije dopalo. Povratne informacije čitalaca su nam važne da bismo kreirali naslove od kojih ćete dobiti maksimum.

Da biste nam poslali povratne informacije, jednostavno nam pošaljite e-mail na adresu informatori@kombib.rs i u naslovu poruke napišite naslov knjige.

# PREUZIMANJE PRIMERA KODA

Možete da preuzmete fajlove sa primerima koda prateći sledeće korake:

1. Posetite veb stranicu knjige razvoj Python mikroservisa: https://goo.gl/6xynL9
2. Kliknite Preuzmite kod: https://goo.gl/RVkd21

Kada su fajlovi preuzeti, ekstrahujte direktorijum, koristeći najnoviju verziju:

- WinRAR / 7-Zip za Windows
- Zipeg / iZip / UnRarX za Mac
- 7-Zip / PeaZip za Linux

# ŠTAMPARSKE GREŠKE

Iako smo preduzeli sve mere da bismo obezbedili tačnost sadržaja, greške su moguće. Ako pronađete grešku u nekoj od naših knjiga (u tekstu ili u kodu), bili bismo zahvalni ako biste nam to prijavili. Na taj način možete da poštedite druge čitaoce od frustracije, a nama da pomognete da poboljšamo naredne verzije ove knjige. Ako pronađete neku štamparsku grešku, molimo vas da nas obavestite, tako što ćete posetiti stranicu http://ww w.packtpub.com/submit – errata, selektovati knjigu, kliknuti na link Errata Submission Form i uneti detalje o grešci koju ste pronašli. Kada je greška verifikovana, vaša prijava će biti prihvaćena i greška će biti aploudovana na naš veb sajt ili dodata u listu postojećih grešaka, pod odeljkom Errata za određeni naslov.

Da biste pogledali prethodno prijavljene greške, posetite stranicu https://www.packtpub.com/books/content/support i unesite naslov knjige u polje za pretragu. Tražena informacija će biti prikazana u odeljku Errata.

# PIRATERIJA

Piraterija autorskog materijala na Internetu je aktuelan problem na svim medijima. Mi u „Packtu“ zaštitu autorskih prava i licenci shvatamo veoma ozbiljno. Ako pronađete ilegalnu kopiju naših knjiga, u bilo kojoj formi, na Internetu, molimo vas da nas o tome obavestite i pošaljete nam adresu lokacije ili naziv web sajta da bismo mogli da podnesemo tužbu.

Kontaktirajte sa nama na adresi copyright@packtpub.com i informatori@kombib.rs pošaljite nam link ka sumnjivom materijalu.

# UVOD

U kompaniji „Mozilla“ je pre sedam godina, kada sam se zaposlio u njoj, započelo pisanje veb servisa za neke funkcije Firefoxa. Neki od njih su na kraju postali mikroservisi. Ova promena se nije desila odjednom, već postepeno. Prvi pokretač ove promene je bila činjenica sa smo prebacili sve servise na cloud provajder i počeli da vršimo interakciju sa nekim nezavisnim servisima. Kada hostujemo aplikacije u cloudu, arhitektura mikroservisa postaje prirodno prilagođena. Drugi pokretač je bio Firefox Account projekat. Želeli smo da ponudimo mogućnost da korisnici imaju jedan identitet za vršenje interakcije sa servisima iz Firefoxa. Trebalo je da svi naši servisi vrše interakciju sa istim provajderom identiteta i neki delovi na strani servera su počeli da se redizajniraju kao mikroservisi da bi postali efikasniji u tom kontekstu.

Mislim da je mnogo veb programera prošlo kroz isto ovo iskustvo ili sada prolazi. Takođe verujem da je Python jedan od najboljih jezika za pisanje malih i efikasnih mikroservisa; njegov ekosistem je živahan i najnovije funkcije Pythona 3 čine Python konkurentnim u toj oblasti, nasuprot Node.jsa, koji je u poslednjih pet godina imao zavidan rast.

Tome je posvećena ova knjiga; ja sam želeo da podelim svoje iskustvo u pisanju mikroservisa u Pythonu kroz primere upotrebe koje sam kreirao za ovaj cilj – za Runnerly, koji je dostupan na GitHubu, pa možete malo da ga istražite. Možete da kontaktirate sa mnom pomoću GitHuba, da ukažete na greške ako ih primetite i možemo zajedno nastaviti da učimo pisanje odličnih Python aplikacija.

# 1

# Razumevanje mikroservisa

Mi programeri uvek pokušavamo da poboljšamo način izgradnje softvera i od ere bušenih kartica mnogo smo napredovali.

Trend mikroservisa je poboljšanje koje je stiglo u poslednjih nekoliko godina, delimično na osnovu želja kompanija za ubrzanjem ciklusa izdanja - one žele da isporuče nove proizvode i nove funkcije svojim potrošačima što je moguće brže. Žele da budu *agilne* čestim iteracijama i žele da isporučuju verzije sve brže.

Ako hiljade ili čak milioni potrošača koristi vaš servis, postavljanje proizvodnje eksperimentalne funkcije u uklanjanje te funkcije ako ne funkcioniše smatra se najboljom praksom, umesto čekanja mesecima pre nego što je publikujete.

Pojedine kompanije, kao što je „Netflix“, promovišu svoje tehnike kontinualnog isporučivanja u kojima su male promene veoma često izvršene u proizvodnji i testirane na podskupu korisnika. One su razvile alatke, kao što je Spinnaker (`http://www.spinnaker.io/`), za automatizaciju što više koraka za ažuriranje proizvodnje i isporučivanje njihovih funkcija u cloud kao nezavisnih mikroservisa.

Međutim, ako čitate „Hacker News“ ili „Reddit“, može vam biti prilično teško da razabarete šta vam je korisno, a šta je samo nebitna informacija u novinarskom stilu.

*„Ako vaše novine obećavaju spasenje i učinite ih nečim strukturiranim ili virtuelnim, ili apstraktnim, distribuiranim, ili višim ili aplikativnim, možete biti skoro sigurni da ste pokrenuli novi kult.*

*Edsger W. Dijkstra*

Ovo poglavlje će vam pomoći da razumete šta su mikroservisi, a zatim ćemo se fokusirati na različite načine na koje možete da ih implementirate pomoću Pythona. Poglavlje je sklopljeno od sledećih nekoliko odeljaka:

- Opis servisno-orijentisane arhitekture
- Monilitni pristup izgradnje aplikacije
- Mikroservisni pristup izgradnje aplikacija
- Prednosti mikroservisa
- Zamke u mikroservisima
- Implementiranje mikroservisa pomoću Pythona

Nadam se da ćete do kraja poglavlja moći da „zaronite“ u izgradnju mikroservisa, uz dobro razumevanje šta su oni i šta nisu i kako možete da upotrebite Python.

# POČECI SERVISNO-ORIJENTISANE ARHITEKTURE

Pošto ne postoji zvanični standard za mikroservise, postoji mnoštvo definicija za njih. Pojedinci često pominju servisno-orijentisanu arhitekturu (SOA) kada pokušavaju da objasne šta su mikroservisi.

> *SOA prethodi mikroservisima - njen osnovni princip je ideja da se organizuju aplikacije u diskretne jedinice funkcionalnosti kojima se može daljinski pristupiti da se na njih deluje i koje se nezavisno ažuriraju.*
>
> *Wikipedia*

Svaka jedinica u ovoj prethodnoj definiciji je samostalni servis koji implementira jedan faset posla i obezbeđuje funkcije kroz neki interfejs.

Iako SOA jasno ističe da servisi treba da budu samostalni procesi, nije rečeno koji protokoli treba da se upotrebe za te procese za međusobnu interakciju i ostaje prilično nejasna u pogledu načina raspoređivanja i organizovanja aplikacije.

U **SOA Manifestu** (http://www.soa-manifesto.org), koji je publikovan na Vebu 2009. godine, stručnjaci koji su ga potpisali čak ne spominju da li servisi međusobno vrše interakciju pomoću mreže.

SOA servisi mogu da komuniciraju pomoću **Inter-Process Communicationa (IPC-a)**, koristeći priključke na istoj mašini, korišćenjem deljene memorije i indirektnih redova poruka ili, čak, pomoću **Remote Procedure Callsa (RPC-a)**. Opcije su razne i na kraju SOA može da bude sve i svašta, sve dok ne pokrenete ceo kod aplikacije u jednom procesu.

Međutim, uobičajeno je reći da su mikroservisi specijalizacija SOA ciljeva, koji su počeli da se pojavljuju poslednjih godina, zato što ispunjavaju neke SOA ciljeve, kao što je izgradnja aplikacija pomoću samostalnih komponenata koje međusobno komuniciraju.

Ako želimo da formulišemo kompletnu definiciju šta su mikroservisi, najbolji način je da prvo pogledamo kako je izgrađena većina softvera.

# MONOLITNI PRISTUP

Upotrebićemo jedan veoma jednostavan primer tradicionalne monolitne aplikacije: veb sajt za rezervaciju hotela.

Osim statičnog HTML sadržaja, veb sajt ima funkcije za rezervacije, koje će omogućiti korisnicima da rezervišu hotele u bilo kom gradu na svetu. Korisnici mogu da pretražuju hotele, pa da rezervišu smeštaj u njima, koristeći kreditnu karticu.

Kada korisnik izvrši pretragu na veb sajtu hotela, aplikacija prolazi kroz sledeće korake:

1. Pokreće dva SQL upita za baze podataka hotela.
2. HTTP zahtev za servis partnera je kreiran za dodavanje više hotela u listu.
3. Generisana je stranica sa rezultatima upotrebom mehanizma HTML šablona.

Od te tačke, kada korisnik pronađe odgovarajući hotel i klikne na njega da bi u njemu rezervisao smeštaj, aplikacija izvršava sledeće korake:

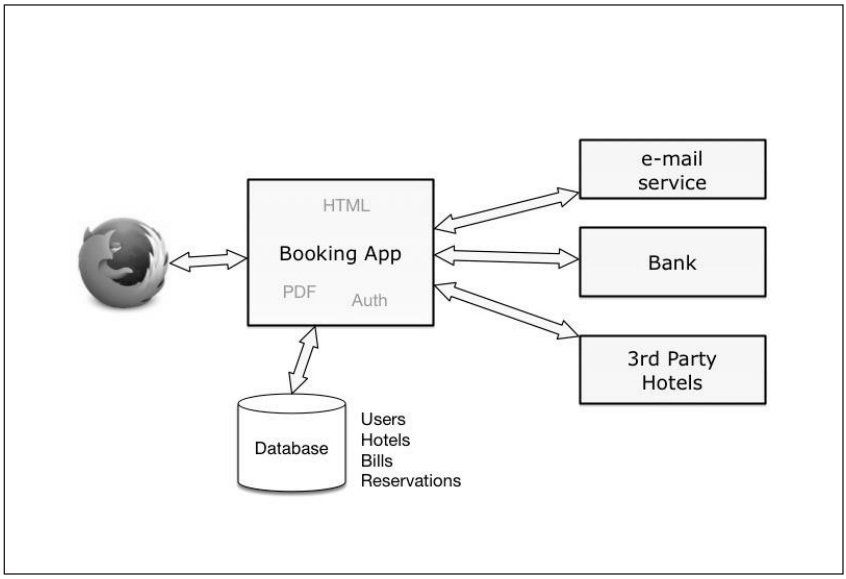
1. Korisnik je kreiran u bazi podataka ako je potrebno i treba da mu se proveri identitet.
2. Plaćanje se izvršava interakcijom sa veb servisom banke.
3. Aplikacija snima detalje o plaćanju u bazi podataka zbog pravnih razloga.
4. Potvrda prijema je generisana pomoću PDF generatora.
5. E-mail rekapitulacije je poslat korisniku pomoću e-mail servisa.
6. E-mail u vezi rezervacije je prosleđen nezavisnom hotelu pomoću e-mail servisa.
7. Unos baze podataka je dodat za praćenje rezervacije.

Ovaj proces je, naravno, pojednostavljeni model, ali je prilično realističan.

Aplikacija vrši interakciju sa bazom podataka koja sadrži informacije o hotelu, detalje o rezervaciji i naplati, informacije o korisniku i tako dalje. Takođe vrši interakciju sa eksternim servisima za slanje e-mailova, izvršavanje plaćanja i dodavanje naziva za više hotela.

U dobroj staroj LAMP (Linux-Apache-MySQL-Perl/PHP/Python) arhitekturi svaki ulazni zahtev generiše kaskadu SQL upita u bazi podataka i nekoliko mrežnih poziva ka eksternim servisima, a zatim server generiše HTML odgovor, koristeći mehanizam šablona.

U sledećem dijagramu ilustrovana je ova centralizovana arhitektura.




Ovo je tipična monolitna aplikacija, koja ima mnogo očiglednih prednosti.

Najveća prednost je što se cela aplikacija nalazi u jednoj osnovi koda, pa, kada započne proces kodiranja projekta, sve postaje jednostavnije. Izgradnja dobrog testa je jednostavna i kod može da se organizuje na jasan i strukturirani način unutar njegove osnove. Skladištenje svih podataka u jednu bazu podataka takođe pojednostavljuje razvoj aplikacije. Mogu da se podešavaju model podataka i način na koji će ih kod zatražiti.

Raspoređivanje je, takođe, jednostavno: možemo da označimo osnovu koda, da izgradimo paket i da ga negde pokrenemo. Da bismo skalirali aplikaciju, možemo da pokrenemo nekoliko instanci aplikacije za rezervacije i nekoliko baza podataka sa postavljenim mehanizmima za repliciranje.

Ako aplikacija ostane mala, ovaj model funkcioniše dobro i jedan tim ga može jednostavno održavati.

Međutim, projekti obično rastu i postaju veći nego što je prvobitno planirano. Ako je cela aplikacija u jednoj osnovi koda, to nam izaziva neke ozbiljne probleme u radu. Na primer, ako treba da izvršimo promene čišćenja koje su velikog obima, kao što je menjanje bankovnog servisa ili sloja baze podataka, cela aplikacija postaje veoma nestabilna. Ove promene su veoma značajne u „životu“ projekta i zahtevaju mnogo dodatnih testiranja za raspoređivanje nove verzije.

Male promene mogu, takođe, generisati kolateralnu štetu, zato što različiti delovi sistema imaju različite zahteve za vreme ispravnog rada i za stabilnost. Postavljanje procesa za naplatu i rezervaciju u rizičnu situaciju, zato što je funkcija koja kreira PDF srušila server, malo je problematično.

Nekontrolisan rast je još jedan problem. Aplikacija dobija nove funkcije i ako programeri napuštaju projekat i priključuju se projektu, organizacija koda može da postane neuredna, a testovi sporiji. Ovaj rast se, obično, završava sa špageti osnovom koda, koja je veoma teška za održavanje, a neuredne baze podataka zahtevaju komplikovane planove migracije uvek kada neki od programera preradi model podataka.

Veliki projekti softvera obično zastarevaju za dve godine, a zatim polako počinju da se pretvaraju u nerazumljivu zbrku, koja je veoma teška za održavanje. A to se ne dešava zato što su programeri loši, već zato što, dok raste kompleksnost, sve manje ljudi razume implikacije veoma malih promena, pa pokušavaju da rade u izolaciji u jednom uglu osnove koda, tako da, kada pogledate prikaz od 10.000 stopa projekta, vidite haos.

I programeri koji rade na takvim projektima sanjaju o izgradnji aplikacije „od nule“ upotrebom najnovijeg radnog okvira i zbog toga stalno upadaju u iste probleme – ponavlja se ista „priča“.

Sledeće tačke rezimiraju prednosti i mane monolitnog pristupa:

- Započinjanje projekta kao monolitnog je jednostavno i verovatno predstavlja najbolji pristup.
- Centralizovana baza podataka pojednostavljuje dizajn i organizaciju podataka.
- Raspoređivanje jedne aplikacije je jednostavno.
- Svaka promena u kodu može da utiče na nevezane funkcije. Kada se nešto ošteti, cela aplikacija može biti oštećena.
- Kako osnova koda raste, teže je da se održava čista i da bude pod kontrolom.

Naravno, postoje načini za izbegavanje nekih od ovde opisanih problema.

Očigledno rešenje je da razdvojimo aplikaciju u posebne delove, čak i ako će se rezultirajući kod i dalje pokretati u jednom procesu. Programeri to rešavaju izgradnjom svojih aplikacija, koristeći eksterne biblioteke i radne okvire. Oni te alatke kreiraju unutar svojih kompanija ili u zajednici **Open Source Software (OSS)**.

Izgradnja veb aplikacije u Pythonu, ako koristimo radni okvir kao što je **Flask**, omogućava da se fokusiramo na poslovnu logiku i pojednostavljuje eksternalizaciju koda u Flask ekstenzije i male Python pakete. A razdvajanje koda u male pakete je često dobra ideja za kontrolisanje rasta aplikacije.

*„Malo je lepo.“*

*UNIX filozofija*

Na primer, PDF generator opisan u aplikaciji za rezervaciju hotela može da bude poseban Python paket koji koristi **Reportlab** i neke šablone za izvršavanje posla.

Postoji mogućnost da ovaj paket bude ponovo upotrebljen u nekim drugim aplikacijama, a možda, čak, i publikovan u **Python Package Index (PyPI)** za zajednicu.

Međutim, i dalje ćemo graditi jednu aplikaciju i neki problemi ostaju u njoj, kao što su nemogućnost različitih skaliranja delova ili bilo koji drugi indirektan problem koji je predstavljen oštećenom zavisnošću.

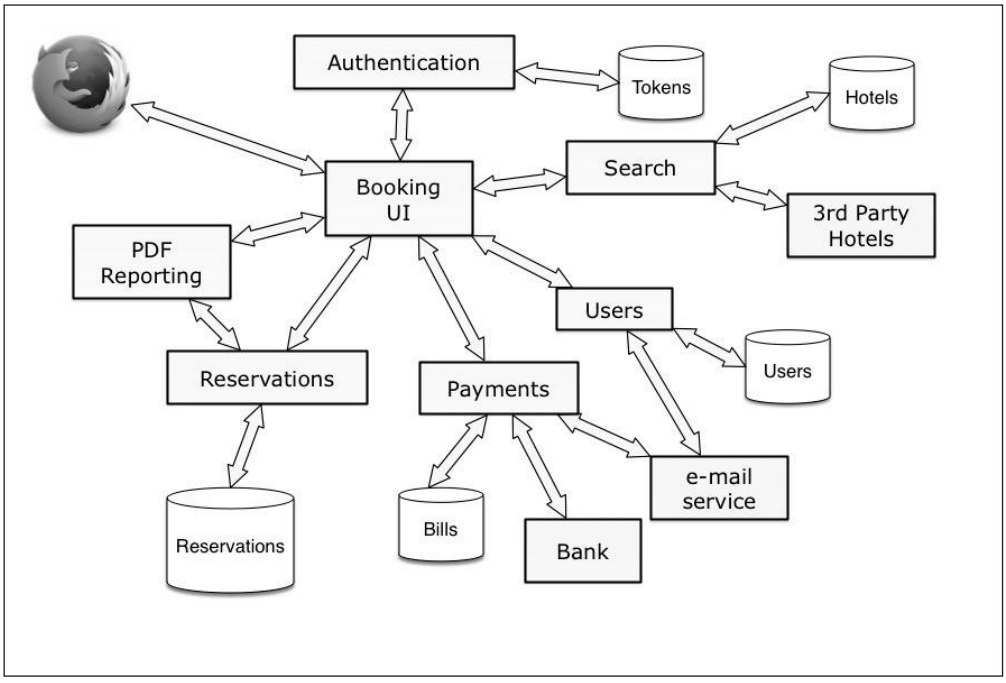
Suočićemo se, čak, i sa novim izazovima, zato što sada koristimo zavisnosti. Jedan od problema je „pakao zavisnosti“. Ako jedan deo aplikacije koristi biblioteku, ali PDF generator može da upotrebi samo specifičnu verziju te biblioteke, postoji mogućnost da će na kraju biti potrebno da se suočimo sa nekim ružnim zaobilaženjima ili, čak, da isključimo zavisnost da bismo rešili problem.

Naravno, svi problemi opisani u ovom odeljku neće se pojaviti prvog dana kada pokrenemo projekat, već će se vremenom gomilati.

Sada ćete da vidite kako bi ista ova aplikacija izgledala ako bismo upotrebili mikroservise za njenu izgradnju.

# MIKROSERVISNI PRISTUP

Ako želimo da izgradimo istu aplikaciju upotrebom mikroservisa, organizovaćemo kod u nekoliko posebnih komponenata koje se pokreću u posebnim procesima. Umesto da imamo jednu aplikaciju koja je za sve odgovorna, razdvojićemo je na više različitih mikroservisa, kao što je prikazano na sledećem dijagramu.




Nemojte dozvoliti da vas zaplaši broj komponenata prikazanih u ovom dijagramu. Interne interakcije monolitne aplikacije su vidljive razdvajanjem delova. Prebacili smo deo složenosti i na kraju smo dobili sledećih sedam samostalnih komponenata:

1. **Booking UI** - servis čeonog prikaza koji generiše veb korisnički interfejs i vrši interakciju sa svim drugim mikroservisima
2. **PDF reporting service** - veoma jednostavan servis koji kreira PDF-ove za potvrde ili bilo koji drugi dokument za određeni šablon ili neke podatke
3. **Search** - servis koji može da bude ispitan za dobijanje liste hotela za određeni naziv grada. Ovaj servis ima sopstvenu bazu podataka.
4. **Payments** - servis koji vrši interakciju sa nezavisnim bankovnim servisom i upravlja bazom podataka naplate. Takođe šalje e-mail o uspešnom plaćanju.
5. **Reservations** - Skladišti rezervacije i generiše PDF-ove.

6. **Users** - Skladišti korisničke informacije i vrši interakciju sa korisnicima pomoću e-maila.
7. **Authentication** - servis zasnovan na OAuthu 2 koji vraća tokene provere identiteta koje svaki mikroservis može da upotrebi za proveru identiteta kada poziva druge.

Ovi mikroservisi, zajedno sa nekoliko eksternih servisa, kao što je e-mail servis, obezbeđuju skup funkcija sličan monolitnoj aplikaciji. U ovom projektu svaka komponenta komunicira upotrebom HTTP protokola, a funkcije su dostupne pomoću RESTful veb servisa.

Ne postoji centralizovana baza podataka, jer svaki mikroservis interno koristi sopstvene strukture podataka, a podaci koji ulaze ili izlaze koriste jezički nezavisan format, kao što je JSON. Mikroservis može da upotrebi i formate XML i YAML, sve dok oni mogu da se proizvede i upotrebe u svakom jeziku i da „putuju“ kroz HTTP zahteve i odgovore.

Servis Booking UI je unekoliko poseban, zato što generiše User Interface (UI). U zavisnosti od radnog okvira čeonog prikaza koji je upotrebljen za izgradnju korisničkog interfejsa, izlaz servisa Booking UI može da bude mešavina HTML-a i JSON-a, ili, čak, čist JSON ako interfejs koristi statičnu alatku zasnovanu na JavaScriptu na strani klijenta za generisanje interfejsa direktno u pretraživaču.

Za razliku od ovog konkretnog slučaja korisničkog interfejsa, veb aplikacija koja je projektovana kao mikroservis je kompozicija od nekoliko mikroservisa koji mogu da vrše interakciju međusobno kroz HTTP da bi obezbedili ceo sistem.

U tom kontekstu mikroservisi su logičke jedinice koje se fokusiraju na veoma određene zadatke. Evo i cele definicije:


Mikroservis je jednostavna aplikacija koja obezbeđuje suženu listu funkcija sa dobro definisanim ugovorom. To je komponenta sa jednom odgovornošću, koja može da bude razvijena i raspoređena samostalno.

Ova definicija ne spominje HTTP ili JSON, zato što možete da razmotrite mali servis zasnovan na UDP-u koji, na primer, razmenjuje binarne podatke kao mikroservis.

Međutim, u našem slučaju u ovoj knjizi svi mikroservisi su samo jednostavne veb aplikacije koje koriste HTTP protokol i koriste i proizvode JSON kada to nije UI.

# PREDNOSTI MIKROSERVISA

Iako arhitektura mikroservisa izgleda komplikovanije nego njen monolitni suparnik, prednosti su višestruke. Ona pruža sledeće:

- razdvajanje dužnosti
- manje projekte za rad
- više opcija za skaliranje i raspoređivanje

## Razdvajanje dužnosti

Pre svega, posebni timovi mogu da nezavisno razviju svaki mikroservis. Na primer, izgradnja servisa za rezervacije može da bude potpuno samostalan projekat. Tim koji je zadužen za ovaj servis može da ga kreira u bilo kom programskom jeziku, koristeći bilo koju bazu podataka, sve dok ima dobro dokumentovan HTTP API.

To takođe znači da je evolucija aplikacije više pod kontrolom, nego u slučaju monolitne aplikacije. Na primer, ako sistem za plaćanje promeni pozadinske akcije sa bankom, uticaj je lokalizovan unutar konkretnog servisa, a ostatak aplikacije ostaje stabilan i, verovatno, netaknut.

Ovo labavo povezivanje poboljšava brzinu projekta, jer se, na nivou servisa, primenjuje filozofija slična principu *jedne odgovornosti*.

Princip jedne odgovornosti je definisao Robert Martin da bi objasnio da treba da postoji samo jedan razlog da se klasa promeni; drugim rečima, svaka klasa treba da obezbedi jednu, dobro definisanu funkciju. Primenjeno na mikroservis, to znači da želimo da se uverimo da se svaki mikroservis fokusira na jednu ulogu.

## Manji projekti

Druga prednost je razdvajanje složenosti projekta. Kada dodamo funkciju u aplikaciju, kao što je pisanje PDF izveštaja, čak i ako je kreirate jasno, osnovni kod će biti veći, komplikovaniji i ponekad sporiji. Izgradnja te funkcije u posebnoj aplikaciji sprečava pojavu ovog problema i olakšava pisanje bilo kojom alatkom. Možemo da je prerađujemo često, da skratimo ciklus izdanja i da ostanemo u toku. Rast aplikacije ostaje pod kontrolom.

Upotreba manjeg projekta takođe smanjuje rizik kada poboljšavamo aplikaciju: ako tim želi da isproba najnoviji programski jezik ili radni okvir, njegovi članovi mogu brzo da kreiraju prototip koji implementira isti API mikroservisa, da ga isprobaju i da odluče da li će da ga zadrže ili ne.

Jedan stvarni primer koji mi je na umu je mikroservis skladišta Firefox Sync. Trenutno postoje neki eksperimenti za prebacivanje sa aktuelne Python + MySQL implementacije na implementaciju zasnovanu na Go serveru, koji skladišti podatke korisnika u samostalne SQLite baze podataka. Taj prototip je u eksperimentalnoj fazi, ali, pošto smo izolovali funkciju skladišta u mikroservis sa dobro definisanim HTTP API-jem, veoma je jednostavno isprobati mali podskup baze korisnika.

# Skaliranje i razvoj

Na kraju, razdvajanje aplikacije na komponente olakšava skaliranje, u zavisnosti od ograničenja. Recimo da počinje da se svakodnevno javlja sve više korisnika koji rezervišu hotele, pa generisanje PDF-a počinje da „zagreva“ CPU-ove. Možete da rasporedite taj konkretan mikroservis na neke servere koji imaju veće CPU-ove.

Još jedan tipičan primer su mikroservisi koji koriste mnogo RAM-a – na primer, oni koji vrše interakciju sa bazama podataka u memoriji, kao što su **Redis** ili **Memcache**. Može da se podesi njihovo raspoređivanje na servere sa manje CPU-a i mnogo više RAM-a.

Prema tome, možemo da rezimiramo prednosti mikroservisa na sledeći način:

- Tim može da razvije svaki mikroservis nezavisno i da upotrebi bilo koji tehnološki stek. On može da definiše prilagođeni ciklus izdanja. Sve što treba da definiše je jezički nezavisan HTTP API.
- Programeri rastavljaju složenost aplikacije na logičke komponente. Svaki mikroservis se fokusira da izvršava dobro jedan zadatak.
- Pošto su mikroservisi samostalne aplikacije, postoji bolja kontrola za raspoređivanje koja olakšava skaliranje.

Arhitektura mikroservisa je korisna u rešavanju mnoštva problema koji se mogu javiti kada aplikacija počne da raste. Međutim, potrebno je da se čuvamo nekih novih problema koje oni uvode u aplikaciju.

# „ZAMKE“ MIKRO SERVISA

Kao što sam ranije napomenuo, izgradnja aplikacije pomoću mikroservisa ima mnogo prednosti, ali nije savršena.

Potrebno je da obratimo pažnju na sledeće probleme sa kojima se možemo suočiti kada kodiramo mikroservise:

- nelogično razdvajanje
- više mrežne interakcije
- skladištenje i deljenje podataka
- problemi kompatibilnosti
- testiranje

Ovi problemi će detaljno biti opisani u sledećim odeljcima.

## Nelogično razdvajanje

Prvi problem arhitekture mikroservisa je kako su oni dizajnirani. Ne postoji način da tim može da kreira savršenu arhitekturu mikroservisa u prvom pokušaju. Neki mikroservisi, kao što je PDF generator, predstavljaju očigledan primer. Međutim, čim se suočimo sa poslovnom logikom, postoji opasnost da će se kod pomeriti pre nego što jasno vidite kako da ga razdvojite u odgovarajući skup mikroservisa.

Projekat treba da „sazri“ u ciklusima isprobavanja-i-neuspeha. A dodavanje i uklanjanje mikroservisa može da bude teže od prerade monolitne aplikacije.

Ovaj problem možemo da ublažimo izbegavanjem razdvajanja aplikacije u mikroservise ako ono nije evidentno.

*Prevremeno razdvajanje je koren svog zla.*

Ako postoji bilo kakva sumnja da razdvajanje ima smisla, zadržavanje koda u istoj aplikaciji je bezbedno. Uvek je lakše razdvojiti kod u novi mikroservis kasnije od spajanja nazad u dva mikroservisa u istoj osnovi koda kada se ispostavi da je odluka bila loša.

Na primer, ako uvek treba da rasporedimo dva mikroservisa zajedno ili ako jedna promena u mikroservisu utiče na model podataka drugog mikroservisa, možda nismo dobro razdvojili aplikaciju i možda bi ta dva servisa trebalo da ponovo budu spojena.

# Više mrežnih interakcija

Drugi problem je količina mrežnih interakcija koje su dodate za izgradnju iste aplikacije. U monolitnoj verziji, čak i ako kod postane neuredan, sve se dešava u istom procesu i možemo da pošaljemo nazad rezultat, bez potrebe da pozivamo previše pozadinskih servisa za izgradnju aktuelnog odgovora.

To zahteva da se obrati pažnja na način kako je pozvan pozadinski servis i nameće pitanja:

- Šta se dešava kada Booking UI ne može da komunicira sa PDF servisom za izveštaje, zbog razdvojene mreže ili laganog servisa?
- Da li Booking UI poziva druge servise sinhrono ili asinhrono?
- Kako će to uticati na vreme odgovora?

Potrebno je da imamo jaku strategiju da bismo mogli da odgovorimo na sva ova pitanja, o čemu će biti reči u Poglavlju 5, Interakcija sa drugim servisima.

# Skladištenje i deljenje podataka

Problematični su i skladištenje i deljenje podataka. Efikasan mikroservis treba da bude nezavisan od drugih mikroservisa i ne bi trebalo da deli bazu podataka. Šta to znači za našu aplikaciju za rezervacije hotela?

To nameće pitanja, kao, na primer:

- Da li koristimo iste korisničke ID-ove u svim bazama podataka ili imamo nezavisne ID-ove u svakom servisu i čuvamo ih kao skrivene detalje implementacije?
- Kada je korisnik dodat u sistem, da li repliciramo neke informacije u drugim bazama podataka servisa pomoću strategija, kao što je „pumpanje“ podataka, ili je to preterivanje?
- Kako da rukujemo uklanjanjem podataka?

Postoje mnogi različiti načini za rešavanje ovih problema, o kojima će biti reči u ovoj knjizi.


Izbegavanje dupliranja podataka što je moguće više dok zadržavamo mikroservise u izolaciji je jedan od najvećih izazova u projektovanju aplikacija zasnovanih na mikroservisima.

# Problemi kompatibilnosti

Još jedan problem se dešava kada promena funkcije utiče na više mikroservisa. Ako promena utiče na nekompatibilan način (u starijim verzijama) na koji podaci „putuju“ između servisa, onda ste u nevolji.

Možemo li da rasporedimo novi servis i da li će on funkcionisati sa starijim verzijama drugih servisa? Ili da li treba da promenimo i rasporedimo nekoliko servisa odjednom? Da li znači da smo pronašli neke servise koji verovatno treba da budu spojeni?

Dobre verzije i API dizajn će nam pomoći da izbegnemo ove probleme, kao što ćemo i otkriti u drugom delu knjige kada budemo gradili našu aplikaciju.

# Testiranje

Na kraju, kada budemo želeli da izvršimo testove s kraja na kraj i da rasporedimo celu aplikaciju, suočićemo se sa mnogo gradivnih blokova. Potrebno je da imamo robustan i agilan proces raspoređivanja da bismo bili efikasni. Potrebno je da se malo poigramo celom aplikacijom kada je razvijamo. Ne možemo u potpunosti da testiramo kod ako imamo samo jedan deo slagalice.

Srećom, postoje mnoge alatke koje olakšavaju raspoređivanje aplikacija koje su građene pomoću nekoliko komponenata. A sve te alatke pomogle su u uspešnosti i prihvatanju mikroservisa i obratno - mikroservisi su pomogli prihvatanje alatki.


Arhitektura stila mikroservisa povećava inovativnost alatki za raspoređivanje i one olakšavaju potvrđivanje arhitektura stila mikroservisa.

Mane upotrebe mikroservisa možemo da rezimiramo ovako:

- Prerano rastavljanje aplikacije na mikroservise može da dovede do arhitekturnih problema.
- Mrežne interakcije između mikroservisa dodaju slabosti i stvaraju dodatne probleme.
- Testiranje i raspoređivanje mikroservisa mogu da budu složeni.
- I najveći izazov – deljenje podataka između mikroservisa je teško.

Za sada, ne bi trebalo previše da brinete o svim manama koje su opisane u ovom odeljku.

Možda vam se čini da su ogromne i možda monolitna aplikacija izgleda kao bolji izbor, ali, na duže staze, razdvajanje projekta u mikroservise će nam olakšati izvršavanje mnoštva zadataka, kao programeru ili kao Operation personu (Opsu).

# IMPLEMENTIRANJE MIKROSERVISA POMOĆU PYTHONA

Python je neverovatno raznovrstan jezik.

Kao što već verovatno znate, koristi se za izgradnju mnoštva različitih vrsta aplikacija – od jednostavnih skriptova sistema, koji izvršavaju zadatke na serveru, do velikih objektno-orijentisanih aplikacija, koje pokreću servise za milione korisnika.

Prema studiji koju je 2014. godine sproveo Philip Guo, a koja je publikovana na veb sajtu **Association for Computing Machinery (ACM)**, Python je premašio jezik Java na vrhunskim univerzitetima u SAD i najpopularniji je jezik za učenje računarskih nauka.

Ovaj trend važi i u softverskoj industriji. Python se nalazi u prvih pet jezika u TIOBE indeksu (http://www.tiobe.com/tiobe-index/) i verovatno je još bolji u svetu veb programiranja, jer se jezici, kao što je C, retko koriste kao glavni jezici za izgradnju veb aplikacija.


U ovoj knjizi mi pretpostavljamo da već poznajete Python programski jezik. Ako niste iskusni Python programer, možete da pročitate knjigu Expert Python Programming (second edition), iz koje ćete naučiti napredne veštine programiranja u Pythonu.

Međutim, neki programeri kritikuju Python da je spor i da nije prilagođen za izgradnju efikasnih veb servisa. Nije sporno da je Python spor. Međutim, ipak je izabrani jezik za izgradnju mikroservisa i mnoge velike kompanije ga rado koriste.

Ovaj odeljak će vam pružiti malo osnove i naučićete različite načine na koje možete da pišete mikroservise pomoću Pythona i steći ćete uvid u asinhrono programiranje, nasuprot sinhronog. Završićemo poglavlje nekim detaljima o performansama Pythona.

Ovaj odeljak se sastoji iz pet delova:

- WSGI standard
- Greenlet i Gevent
- Twisted i Tornado
- asyncio
- performanse jezika

# WSGI standard

Ono što očarava većinu veb programera koji počinju da koriste Python je koliko je jednostavno pokrenuti aplikaciju.

Python veb zajednica je kreirala standard (inspirisan **Common Gateway Interfaceom - CGI-om**) pod nazivom **Web Server Gateway Interface (WSGI)**. Ovaj standard pojednostavljuje umnogome načine na koje možemo da pišemo Python aplikacije da bismo služili HTTP zahteve serveru.

Kad kod koristi ovaj standard, standardni veb serveri, kao što su Apache ili nginx, mogu da izvrše projekat upotrebom WSGI ekstenzija, kao što su uwsgi ili `mod_wsgi`.

Aplikacija treba da obradi ulazne zahteve i pošalje nazad JSON odgovore, a Python uključuje sve te dobre funkcije u svoju standardnu biblioteku.

Možemo da kreiramo potpuno funkcionalni mikroservis koji vraća lokalno vreme servera pomoću osnovnog Python modula, a sadrži manje od 10 linija koda:

```
import json import time

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    return [bytes(json.dumps({'time': time.time()}),
    'utf8')]
```

WSGI protokol je postao osnovni standard i Python veb zajednica ga je dobro prihvatila. Programeri pišu posredničke programe koji su funkcije i koje možemo da priključimo pre ili posle funkcije same WSGI aplikacije da bismo izvršili nešto u okruženju.

Neki veb radni okviri, kao što je **Bottle** (http://bottlepy.org), kreirani su konkretno prema tom standardu i uskoro će svaki radni okvir moći da se upotrebi pomoću WSGI-a na neki način.

Najveći problem WSGI standarda je njegova sinhrona priroda. Funkcija aplikacije koju ste videli u prethodnom kodu je pozvana tačno jednom po ulaznom zahtevu, a kada se funkcija vrati, treba da vrati odgovor. To znači da će funkcija, uvek kada je pozovete, biti blokirana, dok odgovor ne bude spreman.

A pisanje mikroservisa podrazumeva da kod svo vreme treba da čeka odgovore iz različitih mrežnih resursa. Drugim rečima, aplikacija će biti u stanju čekanja i blokiraće klijenta, dok sve ne bude spremno.

To ponašanje je sasvim u redu za HTTP API-je. Mi ne govorimo o izgradnji dvosmernih aplikacija, kao što su one koje su zasnovane na veb priključcima. Međutim, šta se dešava kada imamo nekoliko ulaznih zahteva koji pozivaju aplikaciju u isto vreme?

WSGI serveri će nam omogućiti da pokrenemo rezervne programske niti koje će istovremeno služiti nekoliko zahteva. Međutim, ne možemo pokrenuti hiljade niti i čim se rezervne niti „umore“, sledeći zahtev će blokirati pristup klijenta, čak i ako mikroservis ne radi ništa, već čeka odgovore pozadinskih servisa.

To je jedan od razloga zbog kojeg su radni okviri bez WSGI standarda, kao što su **Twisted** i **Tornado**, a u JavaScript svetu Node.js, postali veoma uspešni – potpuno su asinhroni.

Kada kodiramo Twisted aplikaciju, možemo da upotrebimo povratne pozive za pauziranje u radu i za rezimiranje zadatka za izgradnju odgovora. To znači da možemo da prihvatimo nove zahteve i počnemo da ih obrađujemo. Ovaj model dramatično smanjuje vreme čekanja u procesu. Može da služi hiljade zahteva istovremeno. Naravno, to ne znači da će aplikacija vratiti svaki odgovor brže, već samo da jedan proces može da prihvati više istovremenih zahteva i da „žonglira“ između njih, dok podaci ne budu spremni za slanje nazad.

Ne postoji jednostavan način u WSGI standardu da predstavimo nešto slično; zajednica programera je raspravljala godinama da bi pronašli rešenje, ali neuspešno. Postoji mogućnost da će zajednica programera na kraju upotrebiti WSGI standard za nešto sasvim drugo.

Izgradnja mikroservisa pomoću sinhronih radnih okvira je i dalje moguća i potpuno je u redu ako raspoređivanje koristi ograničenja WSGI standarda jedan zahtev == jedna programska nit.

Međutim, postoji jedan trik za pojačavanje sinhronih veb aplikacija – to je Greenlet, koji je opisan u sledećem odeljku.

# Greenlet i Gevent

Osnovni princip asinhronog programiranja je da proces izvršava nekoliko istovremenih konteksta izvršenja da bi simulirao paralelni rad.

Asinhrone aplikacije koriste petlju događaja koja zaustavlja i rezimira kontekste izvršenja kada je događaj pokrenut – aktivan je samo jedan kontekst i oni se smenjuju. Eksplicitne instrukcije u kodu će ukazati petlji događaja da je ovo mesto na kojem može da privremeno zaustavi izvršenje.

Kada se to desi, proces će potražiti drugi posao na čekanju da bi ga pokrenuo. Na kraju, proces će se vratiti u funkciju i nastaviti gde je zaustavljen. Prebacivanje sa jednog konteksta izvršenja na drugi naziva se komutacija.

**Greenlet** projekat (https://github.com/python-greenlet/greenlet) je paket koji je zasnovan na Stackless projektu, posebnoj CPython implementaciji, i koji obezbeđuje grinlete.

**Grinleti** su pseudoprogramske niti koje su veoma jeftine za instanciranje (za razliku od pravih programskih niti) i koje mogu da se upotrebe za pozivanje Python funkcija. Unutar tih funkcija možemo da se prebacujemo između funkcija i vratimo kontrolu drugoj funkciji.

Komutacija je izvršena pomoću petlje događaja i omogućava da pišemo asinhronu aplikaciju, koristeći paradigmu interfejsa sličnog programskoj niti.

Ovde je primer iz Greenlet dokumentacije:

```
from greenlet import greenlet def test1(x, y):
    z = gr2.switch(x+y)
    print(z)

def test2(u): print (u) gr1.switch(42)

gr1 = greenlet(test1) gr2 = greenlet(test2) gr1.
switch("hello", " world")
```

Dva grinleta u prethodnom primeru se eksplicitno prebacuju sa jednog na drugi.

Za izgradnju mikroservisa na osnovu WSGI standarda, ako interni kod koristi grinlete, možemo da prihvatimo nekoliko paralelnih zahteva i samo se prebacujemo sa jednog na drugi kada znamo da će poziv blokirati zahtev – kao I/O zahtevi.

Međutim, prebacivanje sa jednog grinleta na drugi treba da bude izvršeno eksplicitno i kod koji se dobije može brzo da postane neuredan i težak za razumevanje. U toj situaciji Gevent može postati veoma koristan.

**Gevent** projekat (http://www.gevent.org/) je nadgradnja Greenleta, koja obezbeđuje implicitni i automatski način prebacivanja između grinleta, između ostalog. Obezbeđuje kooperativnu verziju socket modula koji koristi grinlete za automatsko zaustavljanje i rezimiranje izvršenja kada su neki podaci dostupni u priključku. Postoji čak i monkey patch funkcija, koja automatski zamenjuje standardni priključak biblioteke Geventovom verzijom. To čini standardni sinhroni kod „magično“ asinhronim svakog puta kada koristi priključak – pomoću samo jedne dodatne linije:

```
from gevent import monkey; monkey.patch_all()

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    # ...do something with sockets here... return result
```

Ova implicitna „magija“ ima svoju cenu. Da bi Gevent dobro funkcionisao, ceo interni kod treba da bude kompatibilan sa „krpljenjem“ koje izvršava Gevent. Neki paketi iz zajednice će nastaviti da blokiraju kod ili, čak, zbog toga imaju neočekivane rezultate – pogotovo ako koriste C ekstenziju i prosleđuju neke funkcije standardne biblioteke koju je dodao Gevent.

Međutim, u većini slučajeva sve dobro funkcioniše. Projekti koji dobro sarađuju sa Geventom se nazivaju zeleni; kada biblioteka ne funkcioniše dobro, zajednica programera traži od autora da je „učini zelenom“, što se obično i desi.

Eto to je upotrebljeno za skaliranje Firefox Sync servisa u kompaniji „Mozilla“, na primer.

# Twisted i Tornado

Ako gradimo mikroservise u kojima je važno povećavanje broja paralelnih zahteva, veliko je iskušenje isključivanje WSGI standarda i upotreba samo asinhronog radnog okvira, kao što su **Tornado** (http://www.tornadoweb.org/) ili **Twisted** (https://twistedmatrix.com/trac/).

**Twisted** postoji već veoma dugo. Da bismo implementirali isti mikroservis, potrebno je da napišemo malo opširniji kod, kao što je sledeći:

```
import time import json
from twisted.web import server, resource
from twisted.internet import reactor, endpoints

class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
        request.responseHeaders.addRawHeader(b"content-type",
                                             b"application/
                                             json")
        return bytes(json.dumps({'time': time.time()}),
        'utf8')

    site = server.Site(Simple())
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
    endpoint.listen(site)
    reactor.run()
```

Iako je Twisted ekstremno robustan i efikasan radni okvir, postoji nekoliko problema kada gradimo HTTP mikroservise:

- Potrebno je da implementiramo svaku krajnju tačku u mikroservis pomoću klase izvedene iz klase `Resource` da bismo, na taj način, implementirali svaki podržani metod. Za nekoliko jednostavnih API-ja to dodaje puno šablonskog koda.
- Twisted kod može da bude težak za razumevanje i ispravljanje grešaka, zbog njegove asinhrone prirode.
- Veoma je lako upasti u „pakao povratnih poziva“ kada ulančamo previše funkcija koje se pokreću uzastopno jedna nakon druge i kod postaje neuredan.
- Pravilno testiranje Twisted aplikacije je veoma teško i potrebno je da upotrebimo model testiranje koda specifičan za Twisted.

Tornado je zasnovan na sličnom modelu, ali bolje izvršava posao u nekim područjima. Ima jednostavniji sistem za usmeravanje i izvršava sve što je moguće da bi kod učinio bližim čistom Pythonu. Tornado takođe koristi model povratnog poziva, pa ispravljanje grešaka može biti teško.

Međutim, oba radna okvira rešavaju problem praznine da bi se oslonili na nove asinhrone funkcije, predstavljene u Pythonu 3.

## asyncio

Kada je Guido van Rossum počeo da traži rešenja za dodavanje asinhronih funkcija u Python 3, deo zajednice programera forsirao je rešenje slično Geventu, zato što ima puno smisla pisati aplikacije na asinhroni, sekvencijalan način, umesto da se dodaju eksplicitni povratni pozivi, kao u Tornado ili Twisted radnim okvirima.

Međutim, Guido van Rossum je izabrao eksplicitnu tehniku i eksperimentisao je u projektu **Tulip**, inspirisan Twistedom. Na kraju, **asyncio** modul je „rođen“ iz tog sporednog projekta i dodat je u Python.

Retrospektivno, implementiranje mehanizma eksplicitne petlje događaja u Python umesto upotrebe Geventa ima mnogo više smisla. Način na koji su glavni programeri Pythona kodirali asyncio i način na koji su elegantno proširili jezik pomoću ključnih reči async i await za implementiranje korutina učinili su da asinhrone aplikacije građene osnovnim Python 3.5+ kodom izgledaju veoma elegantno i blisko sinhronom programiranju.


Korutine su funkcije koje mogu da zaustave i ponovo pokrenu svoje izvršenje. U Poglavlju 12, Šta dalje?, objašnjeno je detaljno kako su korutine implementirane u Python i kako se upotrebljavaju.

Python je odlično izbegao nered sintakse povratnog poziva, koji ponekad viđamo u Node.js ili Twisted (Python 2) aplikacijama.

Osim korutina, Python 3 je predstavio kompletan skup funkcija i pomoćnih funkcija u paketu asyncio za izgradnju asinhronih aplikacija (pogledajte stranicu https://docs.python.org/3/library/asyncio.html).

Python jezik je sada izrazit jezik, kao što je **Lua**, za kreiranje aplikacija zasnovanih na korutinama. Postoji nekoliko novih radnih okvira koji su prihvatili te funkcije i funkcionisaće samo u Pythonu 3.5+ da bi iskoristili nove funkcije.

KeepSafe platforma ima **aiohttp** radni okvir (http://aiohttp.readthedocs.io), koji je jedan od tih radnih okvira i izgradnja istog mikroservisa, potpuno asinhronog, pomoću njega zahteva samo sledećih nekoliko elegantnih linija:

```
from
aiohttp
import web
import time

async def handle(request):
    return web.json_response({'time': time.time()})
if __name__== '___main___':

    app = web.Application()
    app.router.add_get('/', handle)
    web.run_app(app)
```

U ovom malom primeru smo veoma blizu načinu na koji želimo da implementiramo sinhronu aplikaciju. Jedini znak da koristimo async je ključna reč async, koja označava funkciju za obradu kao korutinu.

I upravo je to što će biti upotrebljeno na svakom nivou asinhrone Python aplikacije od sada. Evo još jednog primera upotrebe PostgreSQL biblioteke aiopg iz dokumentacije projekta:

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd

host=127.0.0.1'

async def go():
                pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
                          assert ret == [(1,)]

loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

Pomoću nekoliko prefiksa `async` i `await` funkcija koja izvršava SQL upit i šalje rezultate izgleda slično kao sinhrona funkcija.

Međutim, asinhroni radni okviri i biblioteke zasnovane na Pythonu 3 su još uvek u nastajanju i ako koristimo asyncio ili radni okvir kao što je aiohttp, potrebno je da se pridržavamo određene asinhrone implementacije za svaku funkciju koja nam je potrebna.

Ako treba da upotrebimo biblioteku koja nije asinhrona u kodu, da bismo je upotrebili iz asinhronog koda, treba da obavimo neki dodatni i izazovan posao radi sprečavanja blokiranja petlje događaja.

Ako se mikroservis suočava sa ograničenim brojem resursa, može njime da se upravlja, ali je, verovatno, bezbednije da se držimo sinhronog radnog okvira, umesto da upotrebimo asinhroni. Uživajmo u postojanju ekosistema zastarelih paketa i sačekajmo dok asyncio ekosistem ne postane sofisticiraniji.

Postoje mnogi odlični sinhroni radni okviri za izgradnju mikroservisa pomoću Pythona, kao što su **Bottle**, **Pyramid** sa **Corniceom** i **Flask**.


Postoji dobra šansa da će u drugom izdanju ove knjige biti upotrebljen asinhroni radni okvir. U ovom izdanju koristimo Flast radni okvir, koji je veoma je robustan i „zreo". Međutim, koji god Python veb radni okvir da koristite, treba da uzvršite sve promere iz ove knjige. Razlog za to je činjenica da je većina uključenog koda prilikom izgradnje mikroservisa veoma bliska čistom Pythonu, a radni okvir uglavnom služi za usmeravanje zahteva i da ponudi nekoliko pomoćnih funkcija.

# Performanse jezika

U prethodnim odeljcima smo opisali dva različita načina za pisanje mikroservisa - koju god tehniku da upotrebimo, brzina Pythona direktno utiče na performansu mikroservisa.

Naravno, svi znamo da je Python sporiji od Java ili Go jezika, ali brzina izvršenja nije uvek prioritet. Mikroservis je često tanak sloj koda koji većinu svog „života" provede u čekanju nekih mrežnih odgovora od drugih servisa. Osnovna brzina je, obično, manje važna od toga koliko će se brzo SQL upiti vratiti sa Postgres servera, zato što će za vraćanje upita biti iskorišćena većina vremena provedenog u izgradnji odgovora.

Međutim, želja za što bržom aplikacijom je razumljiva.

Jedna kontroverzna tema u Python zajednici u vezi ubrzavanja jezika je kako Global Interpreter Lock (GIL) muteks može da uništi performanse zato što višenitne aplikacije ne mogu da upotrebe više procesa.

Opravdani su razlozi za postojanje GIL-a. On štiti delove koji nisu bezbedni za niti CPython interpretera i postoji i u drugim jezicima, kao što je Ruby. A svi pokušaji da ga uklonimo do sada nisu bili uspešni za proizvodnju brže CPython implementacije.


Larry Hasting radi na CPython projektu bez GIL-a, čiji je naziv Gilectomy (https://github.com/larryhastings/gilectomy). Minimalni cilj je da se osmisli implementacija bez GIL-a, koja može da se pokreće u jednonitnoj aplikaciji brzo kao i CPython. U vreme pisanja ove knjige ona je i dalje sporija od CPythona. Ali je interesantno pratiti ovaj rad i videti da li će jednog dana dostići istu brzinu. To bi učinilo CPython bez GIL-a veoma pogodnim za upotrebu.

Osim što sprečava upotrebu više jezgara u istom procesu, GIL će, takođe, malo degradirati perfomansu za mikroservise pri velikom opterećenju, zbog poziva sistema koje predstavlja muteks.

Međutim, svi pregledi GIL-a su bili korisni: obavljen je posao u poslednjih nekoliko godina za smanjivanje konkurencije GIL-a u interpreteru, a u nekim područjima performansa Pythona je znatno poboljšana.

Čak i ako osnovni tim ukloni GIL, Python je interpretiran jezik i sakuplja otpatke, pa ima problematičnu performansu zbog tih karakteristika.

Python obezbeđuje modul dis koji pomaže da vidimo kako interpreter rastavlja funkciju. U sledećem primeru interpreter će rastaviti jednostavnu funkciju koja rezultira povećanom vrednošću iz sekvence u manje od 29 koraka:

```
          >>> def myfunc(data):
          ...     for value in data:
          ...         yield value + 1
          ...
          >>> import dis
          >>> dis.dis(myfunc)
2           0 SETUP_LOOP              23 (to 26)
            3 LOAD_FAST                0 (data)
            6 GET_ITER
      >>    7 FOR_ITER                15 (to 25)
            1 0 STORE_FAST             1 (value)

3          13 LOAD_FAST                1 (value)
           16 LOAD_CONST               1 (1)
           19 BINARY_ADD
           20 YIELD_VALUE
           21 POP_TOP
           22 JUMP_ABSOLUTE        7
    >>     25 POP_BLOCK
    >>     26 LOAD_CONST               0 (None)
           29 RETURN_VALUE
```

Slična funkcija napisana u statički kompajliranom jeziku će dramatično smanjiti broj potrebnih operacija za dobijanje istog rezultata. Međutim, postoje načini da ubrzamo izvršenje Pythona.

Jedan način je da napišemo deo koda u kompajliranom kodu izgradnjom C ekstenzije ili upotrebom statične ekstenzije jezika, kao što je Cython (http://cython.org/), ali to će kod učiniti previše komplikovanim.

Još jedno rešenje, koje najviše obećava, je da jednostavno pokrenemo aplikaciju korišćenjem **PyPy** interpretera (http://pypy.org/).

PyPy implementira **Just-In-Time (JIT)** kompajler, koji direktno zamenjuje, u vreme izvršenja, delove Pythona mašinskim kodom koji CPU može direktno da upotrebi. Ceo trik za JIT je da detektuje realno vreme, pre izvršenja, kada i kako da to uradi.

Čak i ako je PyPy uvek nekoliko Python verzija iza CPythona, dostigao je tačku u kojoj možemo da ga upotrebimo u proizvodnji i njegove performanse mogu biti prilično dobre. U jednom od projekata u kompaniji „Mozilla“, koji zahteva brzo izvršenje, PyPy verzija je skoro isto toliko brza kao i Go verzija i odlučili smo da upotrebimo Python.


Pypy Speed Center veb sajt je odlično mesto gde možete da pogledate PyPy upoređen sa CPythonom (http://speed.pypy.org/).

Ako program koristi C ekstenzije, treba da ih rekompajlirate za PyPy, a to može da bude problematično. Posebno, ako drugi programeri održavaju ekstenzije koje mi koristimo.

Međutim, ako gradimo mikroservise koristeći standardni skup biblioteka, postoji mogućnost da će dobro funkcionisati sa PyPy interpreterom, pa vredi isprobati.

U svakom slučaju, za većinu projekata prednosti Pythona i njegovog ekosistema u velikoj meri prevazilaze probleme u vezi sa performansama opisanim u ovom odeljku, zato što dodatni utrošci memorije u mikroservisima retko predstavljaju problem. A kada je performansa problematična, pristup mikroservisa omogućava da ponovo napišemo komponente koje su kritične za performansu, bez uticaja na ostatak sistema.

# REZIME

U ovom poglavlju smo uporedili monolitni pristup, nasuprot mikorservisa, za izgradnju veb aplikacija i postalo je očigledno da nije binarni svet taj u kojem treba da izaberemo model prvog dana i da se njega pridržavamo.

Trebalo bi da posmatramo mikroservise kao poboljašnje aplikacije koja je počela svoj „život“ kao monolitna aplikacija.

Kako projekat „sazreva“, delovi logike servisa treba da se prebace u mikroservise. To je veoma koristan pristup, kao što ste naučili u ovom poglavlju, ali treba da se primeni pažljivo da bismo izbegli upadanje u neke uobičajene „zamke“.

Još jedna važna lekcija je da se Python smatra jednim od najboljih jezika za pisanje veb aplikacija, i prema tome, i mikroservisa – to je izabrani jezik u drugim područjima i zbog toga što obezbeđuje puno starih radnih okvira i paketa za izvršavanje posla.

U ovom poglavlju smo na brzinu pregledali nekoliko radnih okvira, i sinhronih i asinhronih, a u ostatku knjige ćemo koristiti Flask radni okvir.

Python je spor jezik i to može da predstavlja problem u veoma specifičnim slučajevima. Međutim, pošto znamo šta ga čini sporim, i različita rešenja će, obično, biti dovoljna da taj problem bude irelevantan.