



Flask

Web Development

RAZVOJ WEB APLIKACIJA SA PYTHON-om

Miguel Grinberg

Flask Web Development

Take full creative control of your web applications with Flask, the Python-based microframework. With the second edition of this hands-on book, you'll learn Flask from the ground up by developing a complete, real-world application created by author Miguel Grinberg. This refreshed edition accounts for important technology changes that have occurred in the past three years.

Explore the framework's core functionality, and learn how to extend applications with advanced web techniques such as database migrations and an application programming interface. The first part of each chapter provides you with reference and background for the topic in question, while the second part guides you through a hands-on implementation.

If you have Python experience, you're ready to take advantage of the creative freedom Flask provides. Three sections include:

- **A thorough introduction to Flask:** explore web application development basics with Flask and an application structure appropriate for medium and large applications
- **Building Flasky:** learn how to build an open source blogging application step-by-step by reusing templates, paginating item lists, and working with rich text
- **Going the last mile:** dive into unit testing strategies, performance analysis techniques, and deployment options for your Flask application

“The second edition upholds the strong tradition of Miguel's blog posts and the book's first edition that, together, fueled my learnings about Flask, including database interactions and deployments.”

—Jason Myers

author, *Essential SQLAlchemy*,
2nd Edition (O'Reilly)

Miguel Grinberg has over 25 years of experience as a software engineer. He blogs at <https://blog.miguelgrinberg.com> about a variety of topics including web development, Python, robotics, photography, and the occasional movie review.

US \$44.99

CAN \$59.99

ISBN: 978-1-491-99173-2



5 4 4 9 9
9 781491 991732



Twitter: @oreillymedia
facebook.com/oreilly

DRUGO IZDANJE

Flask Web Development

Razvoj web aplikacija sa Pythonom

Miguel Grinberg

Peking • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Flask Web Development

Miguel Grinberg

Copyright © 2018 Miguel Grinberg. Sva prava zadržana.

Štampano u Sjedinjenim Američkim Državama.

Objavio O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly knjige se mogu kupiti za obrazovnu, poslovnu ili promotivnu upotrebu. Online izdanja su takođe dostupna za većinu naslova (<http://oreilly.com/safari>). Za više informacija, kontaktirajte naš korporativni/institucionalni odjel prodaje: 800-998-9938 ili corporate@oreilly.com.

Urednik: Allyson MacDonald

Indeks: Ellen Troutman

Urednica produkcije: Colleen Cole

Dizajner enterijera: David Futato

Urednik kopije: Dwight Ramsey

Dizajner omota: Randy Comer

Lektorika: Rachel Head

Ilustrator: Rebecca Demarest

Mart 2018: Drugo izdanje

Istorija revizija za drugo izdanje

2018-03-02: Prvo izdanje

Pogledajte <http://oreilly.com/catalog/errata.csp?isbn=9781491991732> za detalje o izdanju.

O'Reilly logo je registrovani zaštitni znak kompanije O'Reilly Media, Inc. Flask Web Development, naslovna slika i srodnina robna marka su zaštitni znakovi O'Reilly Media, Inc.

İako su izdavač i autor uložili napore u dobroj vjeri kako bi osigurali da su informacije i upute sadržane u ovom djelu tačne, izdavač i autor se odriču svake odgovornosti za greške ili propuste, uključujući bez ograničenja odgovornost za štetu nastalu upotrebom ili oslanjanje na ovaj rad. Korištenje informacija i uputstava sadržanih u ovom radu je na vlastitu odgovornost. Ako bilo koji primjer koda ili druga tehnologija koju ovo djelo sadrži ili opisuje podliježe licencama otvorenog koda ili pravima intelektualnog vlasništva drugih, vaša je odgovornost osigurati da je vaše korištenje u skladu sa takvim licencama i/ili pravima.

978-1-491-99173-2

[LSI]

Sadržaj

Predgovor.....	xii
----------------	-----

Dio I. Uvod u Flask

1. Instalacija.....	1
Kreiranje virtuelnog okruženja direktorijuma aplikacija	2
	2
Kreiranje virtuelnog okruženja sa Python 3	3
Kreiranje virtuelnog okruženja sa Python 2	3
Rad sa virtuelnim okruženjem	4
Instaliranje Python paketa sa pip	5
2. Osnovna struktura aplikacije.....	7
Inicijalizacija	7
Rute i funkcije pregleda	8
Kompletna aplikacija	9
Web server za razvoj	10
Dynamic Routes	12
Režim za otklanjanje grešaka	13
Opcije komandne linije	15
Ciklus zahtjev-odgovor	17
Konteksti aplikacije i zahtjeva	17
Zahtjev za otpremu	18
Objekat zahtjeva	19
Request Hooks	20
Odgovori	21
Flask Extensions	23

3. Šabloni.....	25	26
Jinja2 Template Engine		26
Rendering Templates		27
Vrijednosti		28
Kontrolne strukture		28
Bootstrap integracija sa Flask-Bootstrapom		30
Stranice prilagođenih grešaka		33
Linkovi		36
Static Files		37
Lokalizacija datuma i vremena sa Flask-Moment		38
4. Web obrasci.....		43
Konfiguracija		44
Form Classes		44
HTML prikazivanje obrazaca		47
Rukovanje obrascima u funkcijama prikaza		48
Preusmjeravanja i korisničke sesije		51
Poruka treperi		53
5. Baze podataka.....		57
SQL baze podataka		57
NoSQL baze podataka		58
SQL ili NoSQL?		59
Python Database Frameworks		59
Upravljanje bazom podataka sa Flask-SQLAlchemy Model		61
Definicija Relacije Operacije baze podataka Kreiranje tabela		62
Umetanje redova Modifikacija redova Brisanje redova		64
Postavljanje upita redova		66
		66
		68
		68
		68
Upotreba baze podataka u funkcijama prikaza		71
Integracija sa Python Shell-om		72
Migracije baze podataka sa Flask-Migrate		73
Kreiranje migracionog spremišta		73
Kreiranje skripte za migraciju		74
Nadogradnja baze podataka		75
Dodavanje više migracija		76

6. Email.....	79
Podrška putem e-pošte sa Flask-Mail	79
Slanje e-pošte iz Python Shell-a	81
Integracija e-pošte sa aplikacijom	81
Slanje asinhronne e-pošte	83
7. Velika struktura aplikacije.....	85
Opcije konfiguracije strukture projekta Paket aplikacije korištenjem tvornice aplikacija	85
Implementacija funkcionalnosti aplikacije u nacrtu	86
88	88
88	90
Application Script	93
Requirements File	93
Jedinični testovi	94
Podešavanje baze podataka	96
Pokretanje aplikacije	97

Dio II. Primjer: Aplikacija za društveni blog

8. Autentifikacija korisnika.....	101
Proširenja za provjeru autentičnosti za Flask Sigurnost lozinke Heširanje lozinki sa Werkzeugom Kreiranje nacrta autentikacije Autentifikacija korisnika sa Flask-Login	101
Priprema korisničkog modela za prijave Zaštita ruta Dodavanje obrasca za prijavu	102
Potpisivanje korisnika Upisivanje korisnika Ne razumijevanje kako Flask-Login funkcioniра	105
Testiranje prijava Registracija novih korisnika Dodavanje obrasca za registraciju korisnika	107
Registracija novih korisnika	107
108	108
109	109
111	111
112	112
113	113
114	114
115	115
115	115
117	117
Potvrda računa	118
Generiranje tokena potvrde sa svojim opasnim	118
Slanje e-pošte za potvrdu	120
Upravljanje računa	125

9. Uloge korisnika.....	127	Baza podataka
Reprezentacija uloga	127	Dodjela uloga
Verifikacija uloge	131	
	132	
10. User Proles.....	137	
Informacije o profilu	137	
Stranica korisničkog profila	138	
Profil Editor	141	
Uređivač profila na nivou korisnika	141	
Uređivač profila na nivou administratora	143	
User Avatars	146	
11. Blog Postovi.....	151	Podnošenje i
pričak objave na blogu	151	postova na blogu na stranicama profila
Kreiranje lažnih podataka o objavama na blogu	154	PageRanking dugih lista blog postova
Renderiranje podataka na stranicama	155	Dodavanje
widgeta za paginaciju	155	
	157	
	158	
Objave obogaćenog teksta sa Markdown i Flask-PageDown	161	
Korištenje Flask-PageDown	162	
Rukovanje obogaćenim tekstom na serveru	164	
Trajne veze do postova na blogu	165	
Urednik blogova	167	
12. Sljedbenici.....	171	
Odnosi baze podataka	171	Revidirani odnosi
mnogo-na-mnogih	172	Samoreferencijalni
odnosi Napredni odnosi	174	mnogo-na-
mnogo Pratioci na stranici profila	174	Upitivanje
praćenih postova pomoću baze podataka	178	Pridruživanje
Prikaz praćenih postova na početnoj stranici	181	
	183	
13. Komentari korisnika.....	189	Baza podataka
Reprezentacija komentara	189	Podnošenje komentara i pričak moderiranja komentara
	191	
	193	
14. Sučelja za programiranje aplikacija.....	199	
Uvod u REST	199	

Resursi su sve	200
Request Methods	201
Tijela zahtjeva i odgovora	201
Versioniranje	202
RESTful Web usluge sa Flaskom	203
Kreiranje API nacrta	203
Error Handling	204
Autentifikacija korisnika pomoću Flask-HTTPAuth	206
Autentifikacija zasnovana na tokenima	208
Serijalizacija resursa u i iz JSON-a	210
Implementacija krajnjih tačaka resursa	213
Paginacija velikih kolekcija resursa	216
Testiranje web servisa sa HTTPie-om	217

Dio III. Poslednja milja

15. Testiranje	221
Pribavljanje izvještaja o pokrivenosti koda	221
Flask Test Klijentsko testiranje web	224
aplikacija Testiranje web servisa	225
Sveobuhvatno testiranje sa selenom	228
Da li se isplati?	230
	234
16. Performanse.	237
Zapisivanje Spore performanse baze podataka	237
Profiliranje izvornog koda	239
17. Raspoređivanje.	241
Tok rada implementacije Evidentiranje grešaka tokom implementacije proizvodnog oblaka Heroku platforma	241
242	242
243	243
244	244
Priprema aplikacije	244
Testiranje uz Heroku Local	253
Postavljanje sa git push	254
Postavljanje nadogradnje	255
Docker kontejneri	256
Instaliranje Dockera	256
Izgradnja slike kontejnera	257
Pokretanje kontejnera	261

Provjera radnog kontejnera	262
Guranje slike vašeg kontejnera u vanjski registar	263
Korištenje vanjske baze podataka	264
Orkestracija kontejnera sa Docker Compose	265
Čišćenje starih kontejnera i slika	269
Korištenje Dockera u proizvodnji	270
Tradicionalne implementacije	270
Podešavanje servera	271
Uvoz varijabli okruženja	271
Postavljanje evidencije	272
18. Dodatni resursi	275
Korištenje integriranog razvojnog okruženja (IDE)	275
Pronalaženje flask ekstenzija	276
Dobivanje pomoći	276
Uključivanje u Flask	277
Indeks	279

Predgovor

Flask se izdvaja od ostalih okvira jer omogućava programerima da zauzmu vozačko mjesto i imaju potpunu kreativnu kontrolu nad svojim aplikacijama. Možda ste već čuli frazu „borba protiv okvira“. Ovo se dešava sa većinom okvira kada odlučite da rešite problem sa rešenjem koje nije zvanično. Može biti da želite da koristite drugačiji mehanizam baze podataka, ili možda drugačiji metod autentifikacije korisnika. Odstupanje od putanje koje su postavili programeri okvira zadat će vam mnogo glavobolja.

Flask nije takav. Volite li relacijske baze podataka? Odlično. Flask ih sve podržava. Možda više volite NoSQL bazu podataka? Nema problema uopšte. Flask također radi s njima. Želite koristiti svoj vlastiti motor baze podataka? Uopšte vam nije potrebna baza podataka? Još uvijek dobro. Uz Flask možete odabratи komponente svoje aplikacije, ili čak napisati vlastitu ako je to ono što želite. Nema pitanja!

Ključ ove slobode je da je Flask od samog početka dizajniran da bude proširen. Dolazi sa robusnim jezgrom koje uključuje osnovnu funkcionalnost koja je potrebna svim web aplikacijama i očekuje da ostatak bude obezbeđen od strane mnogih ekstenzija trećih strana u ekosistemu – i, naravno, od vas.

U ovoj knjizi predstavljam svoj radni tok za razvoj web aplikacija sa Flaskom. Ne tvrdim da je ovo jedini pravi način za pravljenje aplikacija sa ovim okvirom. Trebali biste shvatiti moje izvore kao preporuke, a ne kao evanđelje.

Većina knjiga o razvoju softvera pruža male i fokusirane primjere koda koji demonstriraju različite karakteristike ciljne tehnologije u izolaciji, ostavljajući "ljepljivi" kod koji je neophodan za transformaciju ovih različitih karakteristika u potpuno funkcionalnu aplikaciju koju čitalac može shvatiti. Ja imam potpuno drugačiji pristup. Svi primjeri koje predstavljam dio su jedne aplikacije koja počinje vrlo jednostavno i proširuje se u svakom sljedećem poglavljju. Ova aplikacija počinje život sa samo nekoliko linija koda i završava kao lijepo predstavljena aplikacija za bloganje i društveno umrežavanje.

cija.

Za koga je ova knjiga

Trebali biste imati određeni nivo iskustva u Python kodiranju da biste maksimalno iskoristili ovu knjigu. Iako knjiga prepostavlja da nema prethodnog Flask znanja, prepostavlja se da su Python koncepti kao što su paketi, moduli, funkcije, dekoratori i objektno orientirano programiranje dobro shvaćeni. Poznavanje izuzetaka i dijagnosticiranja problema iz praćenja steka bit će vrlo korisno.

Dok radite kroz primjere u ovoj knjizi, provest ćete mnogo vremena u komandnoj liniji. Trebali biste se osjećati ugodno koristeći komandnu liniju vašeg operativnog sistema.

Moderne web aplikacije ne mogu izbjegći korištenje HTML-a, CSS-a i JavaScript-a. Primjer aplikacije koji je razvijen u cijeloj knjizi očito koristi ove, ali sama knjiga ne ulazi u mnogo detalja o ovim tehnologijama i načinu na koji se koriste. Određeni stepen poznavanja ovih jezika se preporučuje ako nameravate da razvijete kompletne aplikacije bez pomoći razvijenih tehnika na strani klijenta.

Objavio sam prateću aplikaciju za ovu knjigu kao otvoreni izvor na GitHubu. Iako GitHub omogućava preuzimanje aplikacija kao običnih ZIP ili TAR datoteka, toplo preporučujem da instalirate Git klijent i upoznate se s kontrolom verzija izvornog koda (barem s osnovnim naredbama za kloniranje i provjeru različitih verzija aplikacije direktno iz spremišta). Kratka lista komandi koje će vam trebati prikazana je u ["Kako raditi s primjerom koda"](#) na stranici xiii. Poželjet ćete koristiti kontrolu verzija i za svoje projekte, pa koristite ovu knjigu kao izgovor da naučite Git!

Konačno, ova knjiga nije potpuna i iscrpna referenca na Flask okvir. Većina karakteristika je pokrivena, ali ovu knjigu biste trebali dopuniti [zvaničnom Flask dokumentacijom](#).

Kako je ova knjiga organizovana

Ova knjiga je podijeljena u tri dijela.

[Prvi dio, Uvod u Flask](#), istražuje osnove razvoja web aplikacija sa Flask framework-om i nekim njegovim proširenjima:

- [Poglavlje 1](#) opisuje instalaciju i podešavanje Flask framework-a. •
- [Poglavlje 2](#) ulazi direktno u Flask sa osnovnom aplikacijom. • [Poglavlje 3](#) uvodi upotrebu šablona u Flask aplikacijama. • [Poglavlje 4](#) uvodi web obrasce. • [Poglavlje 5](#) uvodi baze podataka.

- **Poglavlje 6** uvodi podršku putem e-pošte.
- **Poglavlje 7** predstavlja strukturu aplikacije koja je prikladna za srednje i velike aplikacije.

Dio II, primjer: *Aplikacija za društveno bloganje*, gradi Flasky, aplikaciju za bloganje i društvene mreže otvorenog koda koju sam razvio za ovu knjigu:

- **Poglavlje 8** implementira sistem autentifikacije korisnika. •
- **Poglavlje 9** implementira korisničke uloge i dozvole. •
- **Poglavlje 10** implementira stranice profila korisnika. •
- **Poglavlje 11** kreira interfejs za blogovanje. • **Poglavlje 12** implementira sljedbenike. • **Poglavlje 13** implementira komentare korisnika za postove na blogu. • **Poglavlje 14** implementira interfejs za programiranje aplikacije (API).

Dio III, Posljednja milja, opisuje neke važne zadatke koji nisu direktno povezani sa kodiranjem aplikacije koje treba razmotriti prije objavljivanja aplikacije:

- **Poglavlje 15** detaljno opisuje različite strategije testiranja jedinica. •
- **Poglavlje 16** daje pregled tehnika analize performansi. • **Poglavlje 17** opisuje opcije implementacije za Flask aplikacije, uključujući tradicionalne cionalna rješenja zasnovana na oblaku i na kontejnerima.
- **Poglavlje 18** navodi dodatne resurse.

Kako raditi s primjerom koda

Primjeri koda predstavljeni u ovoj knjizi dostupni su za preuzimanje na <https://github.com/miguelgrinberg/flasky>.

Istorijska urezivanja u ovom spremištu je pažljivo kreirana kako bi odgovarala redosledu kojim su koncepti predstavljeni u knjizi. Preporučeni način rada sa kodom je da provjerite urezivanje počevši od najstarijeg, a zatim se krećete naprijed kroz listu urezivanja kako napredujete s knjigom. Kao alternativa, Git-Hub će vam takođe omogućiti da preuzmete svako urezivanje kao ZIP ili TAR fajl.

Ako odlučite koristiti Git za rad sa izvornim kodom, onda morate instalirati Git klijenta, koji možete preuzeti sa <http://git-scm.com>. Sljedeća naredba preuzima primjer koda koristeći Git:

```
$ git klon https://github.com/miguelgrinberg/flasky.git
```

Komanda git clone instalira izvorni kod sa GitHub-a u fasciklu flasky2 koja je kreirana u trenutnom direktorijumu. Ovaj folder ne sadrži samo izvorni kod; kopija Git repozitorija sa cjelokupnom istorijom promjena napravljenih na aplikaciji je također uključena.

U prvom poglavlju od vas će biti zatraženo da pogledate početno izdanje aplikacije, a zatim ćete, na odgovarajućim mjestima, biti upućeni da idete naprijed u historiji.

Git komanda koja vam omogućava kretanje kroz istoriju promjena je git checkout.

Evo primjera:

```
$ git checkout 1a
```

1a referenciran u naredbi je oznaka: imenovana tačka u historiji urezivanja projekta. Ovo spremište je označeno prema poglavljima knjige, tako da oznaka 1a korištena u primjeru postavlja datoteke aplikacije na početnu verziju korištenu u Poglavlju 1. Većina [poglavlja](#) ima više od jedne oznake povezane s njima, tako da npr., oznake 5a, 5b i tako dalje su inkrementalne verzije predstavljene u [poglavlju 5](#).

Kada pokrenete git checkout komandu kao što je upravo prikazano, Git će prikazati poruku upozorenja koja vas obavještava da ste u stanju „odspojene GLAVE“. To znači da niste ni u jednoj specifičnoj grani koda koja može prihvati nova urezivanja, već umjesto toga gledate određeno urezivanje usred historije promjena projekta.

Nema razloga da budete uznemireni ovom porukom, ali treba da imate na umu da ako izvršite modifikacije bilo koje datoteke dok su u ovom stanju, izdavanje još jednog git checkout - a neće uspeti, jer Git neće znati šta da radi sa promenama napravio si.

Dakle, da biste mogli da nastavite da radite sa projektom, moraćete da vratite datoteke koje ste promenili u prvobitno stanje. Najlakši način da to uradite je pomoću git reset komande:

```
$ git reset --hard
```

Ova naredba će uništiti sve lokalne promjene koje ste napravili, tako da biste trebali sačuvati sve što ne želite da izgubite prije nego što koristite ovu naredbu.

Pored provjere izvornih datoteka za verziju aplikacije, u određenim trenucima morat ćete izvršiti dodatne zadatke postavljanja. Na primjer, u nekim slučajevima morat ćete instalirati nove Python pakete ili primijeniti ažuriranja na bazu podataka. Biće vam rečeno kada su to neophodne.

S vremenom na vreme, možda ćete želeti da osvežite svoje lokalno spremište sa onog na GitHubu, gde su možda primjenjene ispravke grešaka i poboljšanja. Komande koje to postižu su:

```
$ git dohvati --sve $  
git dohvati --oznake $  
git reset --hard origin/master
```

Naredbe git fetch se koriste za ažuriranje historije urezivanja i oznaka u vašem lokalnom spremištu sa udaljenog na GitHubu, ali ništa od toga ne utiče na stvarne izvorne datoteke, koje se ažuriraju naredbom git reset koja slijedi.

Još jednom, imajte na umu da ćete svaki put kada se koristi git reset izgubiti sve lokalne promjene koje ste napravili.

Još jedna korisna operacija je da vidite sve razlike između dvije verzije aplikacije. Ovo može biti veoma korisno za detaljno razumijevanje promjene. Iz komandne linije, komanda git diff to može učiniti. Na primjer, da vidite razliku između revizija 2a i 2b, koristite:

```
$ git diff 2a 2b
```

Razlike su prikazane kao zagrpa, što nije baš intuitivan format za pregled promjena ako niste navikli raditi sa datotekama zagrpa. Možda ćete otkriti da su grafička poređenja koja pokazuju GitHub mnogo lakša za čitanje. Na primjer, razlike između revizija 2a i 2b mogu se vidjeti na GitHubu na <https://github.com/miguelgrinberg/flasky/compare/2a...2b>.

Korištenje primjera koda

Ova knjiga je ovdje da vam pomogne da završite svoj posao. Općenito, ako se uz ovu knjigu nudi primjer koda, možete ga koristiti u svojim programima i dokumentaciji. Ne morate nas kontaktirati za dozvolu osim ako ne reproducirate značajan dio koda. Na primjer, pisanje programa koji koristi nekoliko dijelova koda iz ove knjige ne zahtijeva dozvolu. Za prodaju ili distribuciju CD-ROM-a sa primjerima iz O'Reillyjevih knjiga potrebna je dozvola. Odgovaranje na pitanje citiranjem ove knjige i citiranjem primjera koda ne zahtijeva dozvolu. Uključivanje značajne količine primjera koda iz ove knjige u dokumentaciju vašeg proizvoda zahtijeva dozvolu.

Cijenimo, ali ne zahtijevamo, atribuciju. Atribucija obično uključuje naslov, autora, izdavača i ISBN. Na primjer: „Flask Web Development, 2. izdanje, Miguel Grinberg (O'Reilly). Autorska prava 2018. Miguel Grinberg, 978-1-491-99173-2.”

Ako smatrate da je vaša upotreba primjera koda izvan poštene upotrebe ili gore navedene dozvole, slobodno nas kontaktirajte na permissions@oreilly.com.

Konvencije koje se koriste u ovoj knjizi

U ovoj knjizi koriste se sljedeće tipografske konvencije:

Kurziv

Označava nove termine, URL adrese, adrese e-pošte, nazive datoteka i ekstenzije datoteka.

Konstantna širina

Koristi se za izlaz komandne linije i liste programa, kao i unutar paragrafa za upućivanje na komande i programske elemente kao što su imena varijabli ili funkcija, baze podataka, tipovi podataka, varijable okruženja, izjave i ključne riječi.

Konstantna širina

podebljano Prikazuje komande ili drugi tekst koji bi korisnik trebao doslovno otkucati.

Kurzivne ili ugaone zagrade konstantne širine (<>)

Označava tekst koji treba zamijeniti vrijednostima koje je dostavio korisnik ili vrijednostima određenim kontekstom.



Ovaj element označava savjet ili prijedlog.



Ovaj element označava opštu napomenu.



Ovaj element ukazuje na upozorenje ili oprez.

O'Reilly Safari

 **Safari**® (ranije Safari Books Online) je obuka zasnovana na članstvu i referentna platforma za preduzeća, vladu, edukatore i pojedince.

Članovi imaju pristup hiljadama knjiga, video zapisa za obuku, puteva učenja, interaktivnih tutorijala i kuriranih lista za reprodukciju od preko 250 izdavača, uključujući O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press , Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett i Course Technology, između ostalih.

Za više informacija posjetite <http://oreilly.com/safari>.

Kako nas kontaktirati

Molimo da komentare i pitanja u vezi ove knjige uputite izdavaču:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472 800-998-9938 (u
Sjedinjenim Državama ili Kanadi) 707-829-0515
(međunarodno ili lokalno) 707-829-0104 (fax)

Imamo web stranicu za ovu knjigu, na kojoj navodimo greške, primjere i sve dodatne informacije.
Ovoj stranici možete pristupiti na <http://bit.ly/flask-web-dev2>.

Da biste komentarisi ili postavili tehnička pitanja o ovoj knjizi, pošaljite e-poštu na bookquestions@oreilly.com.

Za više informacija o našim knjigama, kursevima, konferencijama i novostima, pogledajte našu web stranicu na <http://www.oreilly.com>.

Pronađite nas na Facebooku: <http://facebook.com/oreilly>

Pratite nas na Twitteru: <http://twitter.com/oreillymedia>

Gledajte nas na YouTube-u: <http://www.youtube.com/oreillymedia>

Priznanja

Ne bih mogao sam napisati ovu knjigu. Dobio sam veliku pomoć od porodice, saradnika, starih prijatelja i novih prijatelja koje sam stekao na tom putu.

Želio bih da se zahvalim Brendanu Kohleru na njegovom detalnjom tehničkom pregledu i na njegovoj pomoći u oblikovanju poglavlja o interfejsima za programiranje aplikacija. Dužni sam i Davidu Baumgoldu, Toddu Brunhoffu, Cecil Rocku i Matthewu Huguesu, koji su pregledali rukopis u različitim fazama dovršetka i dali mi vrlo korisne savjete o tome šta da pokrijem i kako da organizujem materijal.

Pisanje primjera koda za ovu knjigu predstavljalo je značajan napor. Cijenim pomoć Daniela Hofmanna, koji je napravio detaljan pregled koda aplikacije i ukazao na nekoliko poboljšanja. Takođe sam zahvalan svom sinu tinejdžeru, Dilanu Grinbergu, koji je suspendovao svoju zavisnost od Minecraft-a na nekoliko vikenda i pomogao mi da testiram kod na nekoliko platformi.

O'Reilly ima divan program pod nazivom Early Release koji omogućava nestrpljivim čitaocima da imaju pristup knjigama dok se pišu. Neka od mojih čitanja o ranom puštanju-

Oni su otišli dalje i uključili se u korisne razgovore o svom iskustvu rada na knjizi, što je dovelo do značajnih poboljšanja. Želeo bih da se posebno zahvalim Sundipu Gupta, Denu Kerunu, Brajanu Vistiju i Kodiju Skotu za doprinos koji su dali ovoj knjizi.

Osoblje O'Reilly Media je uvijek bilo tu za mene. Iznad svega želim da odam priznanje mojoj divnoj urednici, Meghan Blanchette, za njenu podršku, savjete i pomoć od prvog dana našeg susreta. Meg je iskustvo pisanja moje prve knjige učinila nezaboravnim.

Za kraj, želio bih da se zahvalim odličnoj Flask zajednici.

Dodatne hvala za drugo izdanje

Želio bih da se zahvalim Ally MacDonald, mojoj urednici za drugo izdanje ove knjige, kao i Susan Conant, Rachel Roumeliotis i cijelom timu u O'Reilly Media na njihovoj kontinuiranoj podršci.

Tehnički recenzenti za ovo izdanje odradili su divan posao ukazujući na područja koja treba poboljšati i pružajući mi nove perspektive. Željela bih odati priznanje Loreni Mesa, Diane Chen i Jesse Smith za njihov veliki doprinos kroz njihove povratne informacije i prijedloge. Također cijenim pomoć mog sina, Dilana Grinberga, koji je mukotrpno testirao sve primjere koda.

DIO I

Uvod u Flask

POGLAVLJE 1

Instalacija

Flask je mali okvir prema većini standarda - dovoljno mali da se može nazvati "mikro okvir" i dovoljno mali da ćete, kada se upoznate s njim, vjerovatno moći pročitati i razumjeti sav njegov izvorni kod.

Ali to što je mali ne znači da radi manje od drugih okvira. Flask je dizajniran kao proširivi okvir od temelja; pruža solidnu jezgru sa osnovnim uslugama, dok ekstenzije pružaju ostalo. Budući da možete birati i birati pakete proširenja koje želite, na kraju ćete dobiti lean stack koji nema naduvanost i radi upravo ono što vam je potrebno.

Flask ima tri glavne zavisnosti. Podsistemi za rutiranje, otklanjanje grešaka i interfejs mrežnog prolaza servera (WSGI) dolaze od [Werkzeuga](#); podršku za šablone pruža [Jinja2](#); a integracija komandne linije dolazi iz [Click](#). Autor svih ovih zavisnosti je Armin Ronacher, autor Flaska.

Flask nema izvornu podršku za pristup bazama podataka, provjeru valjanosti web obrazaca, autentifikaciju korisnika ili druge zadatke visokog nivoa. Ove i mnoge druge ključne usluge koje su većini web aplikacija potrebne dostupne su preko ekstenzija koje se integrišu sa osnovnim paketima. Kao programer, imate moći da odaberete ekstenzije koje najbolje rade za vaš projekat, ili čak da napišete svoje ako želite. Ovo je u suprotnosti sa većim okvirom, gdje je većina izbora napravljena umjesto vas i teško ih je ili ponekad nemoguće promijeniti.

U ovom poglavlju ćete naučiti kako instalirati Flask. Jedini uslov je računar sa instaliranim Python-om.



Primeri koda u ovoj knjizi verifikovani su da rade sa Pythonom 3.5 i 3.6. Po želji se može koristiti i Python 2.7, ali s obzirom na to da ova verzija Pythona neće biti održavana nakon 2020. godine, toplo se preporučuje da koristite 3.x verzije.



Ako planirate da koristite Microsoft Windows računar za rad sa primerom koda, morate da odlučite da li želite da koristite „nativni“ pristup zasnovan na Windows alatima ili da podesite računar na način koji vam omogućava da usvojite više mainstream Unix-bazirani skup alata. Kod predstavljen u ovoj knjizi uglavnom je kompatibilan sa oba pristupa. U nekoliko slučajeva u kojima se pristupi razlikuju, slijedi se Unix rješenje i navode se alternative za Windows.

Ako odlučite da pratite Unix radni tok, imate nekoliko opcija. Ako koristite Windows 10, možete omogućiti Windows podsustav za Linux (WSL), što je službeno podržana funkcija koja kreira instalaciju Ubuntu Linuxa koja radi uz izvorni Windows interfejs, dajući vam pristup bash ljudi i kompletan set alata baziranih na Unixu. Ako WSL nije dostupan na vašem sistemu, druga dobra opcija je [Cygwin](#), projekat otvorenog koda koji emulira POSIX podsistem koji koristi Unix i obezbeđuje portove velikog broja Unix alata.

Kreiranje direktorija aplikacija

Za početak, trebate kreirati direktorij u kojem će se nalaziti primjer koda, koji je dostupan u GitHub spremištu. Kao što je objašnjeno u [„Kako raditi s primjerom koda“](#) na stranici xiii, najpogodniji način da to učinite je da provjerite kod direktno sa GitHub-a koristeći Git klijent. Sljedeće naredbe preuzimaju primjer koda sa GitHub-a i inicijaliziraju aplikaciju na verziju 1a, što je početna verzija s kojom ćete raditi:

```
$ git clone https://github.com/miguelgrinberg/flasky.git $ cd flasky $  
git checkout 1a
```

Ako ne želite da koristite Git i umjesto toga ručno otkucate ili kopirate kod, možete jednostavno kreirati prazan direktorij aplikacije na sljedeći način:

```
$ mkdir flasky $  
cd flasky
```

Virtuelna okruženja

Sada kada ste kreirali direktorij aplikacije, vrijeme je da instalirate Flask. Najprikladniji način za to je korištenje virtuelnog okruženja. Virtuelno okruženje

je kopija Python interpretera u koji možete privatno instalirati pakete, bez utjecaja na globalni Python interpreter instaliran u vašem sistemu.

Virtuelna okruženja su veoma korisna jer sprečavaju nered u paketima i sukobe verzija u sistemskom Python interpreteru. Kreiranje virtuelnog okruženja za svaki projekat osigurava da aplikacije imaju pristup samo paketima koje koriste, dok globalni tumač ostaje uredan i čist i služi samo kao izvor iz kojeg se može kreirati više virtuelnih okruženja. Kao dodatna prednost, za razliku od Python tumača za cijeli sistem, virtuelna okruženja se mogu kreirati i njima upravljati bez administratorskih prava.

Kreiranje virtuelnog okruženja sa Python 3

Stvaranje virtuelnih okruženja je oblast u kojoj se tumači Python 3 i Python 2 razlikuju. Uz Python 3, virtuelna okruženja su izvorno podržana venv paketom koji je dio Python standardne biblioteke.



Ako koristite standardni Python 3 interpreter na Ubuntu Linux sistemu, standardni venv paket nije instaliran prema zadanim postavkama. Da ga dodate svom sistemu, instalirajte python3-venv paket na sljedeći način:

```
$ sudo apt-get install python3-venv
```

Komanda koja kreira virtuelno okruženje ima sledeću strukturu:

```
$ python3 -m venv naziv-virtualnog-okruženja
```

Opcija -m venv pokreće venv paket iz standardne biblioteke kao samostalnu skriptu, proslijedjujući željeno ime kao argument.

Sada ćete kreirati virtuelno okruženje unutar flasky direktorijuma. Uobičajena konvencija za virtuelna okruženja je da ih nazovete venv, ali možete koristiti drugo ime ako želite. Provjerite je li vaš trenutni direktorij postavljen na flasky, a zatim pokrenite ovu naredbu:

```
$ python3 -m venv venv
```

Nakon što se naredba završi, imat ćete poddirektorij s imenom venv unutar flasky, sa potpuno novim virtuelnim okruženjem koje sadrži Python interpreter za ekskluzivnu upotrebu u ovom projektu.

Kreiranje virtuelnog okruženja sa Python 2

Python 2 nema venv paket. U ovoj verziji Python interpretera, virtuelna okruženja se kreiraju pomoću pomoćnog programa virtualenv treće strane.

Provjerite je li vaš trenutni direktorij postavljen na flasky, a zatim koristite jednu od sljedeće dvije naredbe, ovisno o vašem operativnom sistemu. Ako koristite Linux ili macOS, naredba je:

```
$ sudo pip install virtualenv
```

Ako koristite Microsoft Windows, obavezno otvorite prozor komandne linije koristeći opciju „Pokreni kao administrator“, a zatim pokrenite ovu naredbu:

```
$ pip install virtualenv
```

Naredba virtualenv uzima ime virtuelnog okruženja kao svoj argument. Provjerite je li vaš trenutni direktorij postavljen na flasky, a zatim pokrenite sljedeću naredbu da kreirate virtualno okruženje pod nazivom venv:

```
$ virtualenv venv  
Novi python izvršni fajl u venv/bin/python2.7 Takođe  
kreiranje izvršnog fajla u venv/bin/python Instalacija  
setuptools, pip, wheel...gotovo.
```

Poddirektorij sa imenom venv će biti kreiran u trenutnom direktorijumu i sve datoteke povezane sa virtuelnim okruženjem će biti unutar njega.

Rad sa virtuelnim okruženjem

Kada želite da počnete da koristite virtuelno okruženje, morate ga "aktivirati". Ako koristite Linux ili macOS računar, virtuelno okruženje možete aktivirati ovom naredbom:

```
$ source venv/bin/aktiviraj
```

Ako koristite Microsoft Windows, komanda za aktivaciju je:

```
$ venv\Scripts\aktivirati
```

Kada je virtuelno okruženje aktivirano, lokacija njegovog Python tumača se dodaje PATH varijabli okruženja u vašoj trenutnoj sesiji komandi, koja određuje gde da tražite izvršne datoteke. Da vas podsjeti da ste aktivirali virtuelno okruženje, naredba za aktivaciju mijenja vašu komandnu liniju tako da uključuje naziv okruženja:

(venv) \$

Nakon što je virtuelno okruženje aktivirano, kucanjem python na komandnoj liniji će se pozvati interpreter iz virtuelnog okruženja umesto tumača za čitav sistem. Ako koristite više od jednog prozora komandne linije, morate aktivirati virtuelno okruženje u svakom od njih.



Dok je aktiviranje virtuelnog okruženja obično najpovoljnija opcija, virtuelno okruženje možete koristiti i bez aktiviranja. Na primjer, možete pokrenuti Python konzolu za venv virtualno okruženje pokretanjem venv/bin/python na Linuxu ili macOS-u ili venv\Scripts\python na Microsoft Windows-u.

Kada završite sa radom sa virtuelnim okruženjem, otkucajte deactivate na komandnoj liniji da vratite varijablu okruženja PATH za vašu terminalsku sesiju i komandni redak u njihova originalna stanja.

Instaliranje Python paketa sa pip

Python paketi se instaliraju sa pip paket menadžerom, koji je uključen u sva virtuelna okruženja. Poput naredbe python , upisivanjem pip u sesiju komandne linije će se pozvati verzija ovog alata koja pripada aktiviranom virtuelnom okruženju.

Da biste instalirali Flask u virtualno okruženje, provjerite je li venv virtualno okruženje aktivirano, a zatim pokrenite sljedeću naredbu:

```
(venv) $ pip install flask
```

Kada izvršite ovu naredbu, pip neće samo instalirati Flask, već i sve njegove zavisnosti. Možete provjeriti koji su paketi instalirani u virtualnom okruženju u bilo kojem trenutku pomoću naredbe pip freeze :

```
(venv) $ pip freeze
click==6.7 Flask==0.12.2
itsdangerous==0.24
Jinja2==2.9.6
MarkupSafe==1.0
Werkzeug==0.12.2
```

Izlaz zamrzavanja pip - a uključuje detaljne brojeve verzija za svaki instalirani paket. Brojevi verzija koje ćete dobiti vjerovatno će se razlikovati od onih prikazanih ovdje.

Također možete provjeriti da li je Flask ispravno instaliran tako što ćete pokrenuti Python interpreter i pokušati ga uvesti:

```
(venv) $ python >>>
import flask
>>>
```

Ako se ne pojave greške, možete sebi čestitati: spremni ste za sljedeće poglavlje, gdje ćete napisati svoju prvu web aplikaciju.

Osnovna struktura aplikacije

U ovom poglavlju ćete naučiti o različitim dijelovima Flask aplikacije. Također ćete napisati i pokrenuti svoju prvu Flask web aplikaciju.

Inicijalizacija

Sve Flask aplikacije moraju kreirati instancu aplikacije. Web server proslijeđuje sve zahtjeve koje primi od klijentata ovom objektu za rukovanje, koristeći protokol koji se zove Web Server Gateway Interface (WSGI, izgovara se "wiz-ghee"). Instanca aplikacije je objekt klase Flask, obično kreirana na sljedeći način:

```
iz flask import Flask
app = Flask(__name__)
```

Jedini potrebnii argument za konstruktor klase Flask je ime glavnog modula ili paketa aplikacije. Za većinu aplikacija, Pythonova varijabla `__name__` je ispravna vrijednost za ovaj argument.



Argument `__name__` koji se prosleđuje konstruktoru aplikacije Flask izvor je zabune među novim Flask programerima. Flask koristi ovaj argument da odredi lokaciju aplikacije, što mu zauzvrat omogućava da locira druge datoteke koje su dio aplikacije, kao što su slike i predlošci.

Kasnije ćete naučiti složenije načine inicijalizacije aplikacije, ali za jednostavne aplikacije to je sve što je potrebno.

Rute i funkcije pregleda

Klijenti kao što su web pretraživači šalju zahtjeve web serveru, koji ih zauzvrat šalje instanci aplikacije Flask. Instanca aplikacije Flask mora znati koji kod treba da pokrene za svaki traženi URL, tako da čuva mapiranje URL-ova u Python funkcije. Povezanost između URL-a i funkcije koja njime rukuje naziva se ruta.

Najprikladniji način za definiranje rute u Flask aplikaciji je kroz app.route dekorator koji je izložen od strane instance aplikacije. Sljedeći primjer pokazuje kako se ruta deklarira pomoću ovog dekoratora:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```



Dekoratori su standardna karakteristika jezika Python. Uobičajena upotreba dekoratora je da registriraju funkcije kao funkcije rukovaoca koje se pozivaju kada se dogode određeni događaji.

Prethodni primjer registrira funkciju index() kao rukovalac za korijenski URL aplikacije. Dok je app.route dekorator poželjna metoda za registraciju funkcija pregleda, Flask također nudi tradicionalniji način za postavljanje ruta aplikacije pomoću metode app.add_url_rule() , koja u svom najosnovnijem obliku ima tri argumenta: URL, naziv krajnje točke i funkcija prikaza. Sljedeći primjer koristi app.add_url_rule() za registraciju funkcije index() koja je ekvivalentna prethodno prikazanoj:

```
def index():
    return '<h1>Zdravo svijete!</h1>'

app.add_url_rule('/', 'index', index)
```

Funkcije kao što je index() koje rukuju URL-ovima aplikacija nazivaju se funkcije prikaza. Ako je aplikacija raspoređena na serveru koji je povezan sa imenom domene www.example.com, onda bi navigacija na http://www.example.com/ u vašem pretraživaču pokrenula index() da se pokrene na serveru. Povratna vrijednost ove funkcije pogleda je odgovor koji klijent prima. Ako je klijent web pretraživač, ovaj odgovor je dokument koji se prikazuje korisniku u prozoru pretraživača. Odgovor koji vraća funkcija prikaza može biti jednostavan niz sa HTML sadržajem, ali može imati i složenije oblike, kao što ćete vidjeti kasnije.



Ugrađivanje nizova odgovora sa HTML kodom u Python izvorne datoteke dovodi do koda koji je teško održavati. Primjeri u ovom poglavlju to čine samo kako bi uveli koncept odgovora. Naučit ćete bolji način generiranja HTML odgovora u 3. poglavlju .

Ako obratite pažnju na to kako se formiraju neki URL-ovi za usluge koje svakodnevno koristite, primijetit ćete da mnogi imaju varijabilne sekcije. Na primjer, URL za vašu Facebook stranicu profila ima format <https://www.facebook.com/<vaše-ime>>, koji uključuje vaše korisničko ime, što ga čini različitim za svakog korisnika. Flask podržava ove vrste URL-ova koristeći posebnu sintaksu u dekoratoru app.route . Sljedeći primjer definira rutu koja ima dinamičku komponentu:

```
@app.route('/user/<name>') def
user(name): return '<h1>Zdravo,
{}</h1>'.format(name)
```

Dio URL-a rute zatvoren u uglastim zagradama je dinamički dio. Svi URL-ovi koji odgovaraju statickim dijelovima bit će mapirani na ovu rutu, a kada se pozove funkcija prikaza, dinamička komponenta će biti proslijeđena kao argument. U prethodnom primjeru, argument name se koristi za generiranje odgovora koji uključuje personalizirani pozdrav.

Dinamičke komponente u rutama su po defaultu nizovi, ali mogu biti i različitih tipova. Na primjer, ruta /user/<int:id> bi odgovarala samo URL-ovima koji imaju cijeli broj u dinamičkom segmentu id , kao što je /user/123. Flask podržava tipove string, int, float i putanja za rute. Tip putanje je poseban tip stringa koji može uključivati kose crte, za razliku od tipa stringa .

Kompletan aplikacija

U prethodnim odjeljcima naučili ste o različitim dijelovima Flask web aplikacije, a sada je vrijeme da napišete svoju prvu. Skripta aplikacije hello.py prikazana u [primjeru 2-1](#) definira instancu aplikacije i jednu rutu i funkciju prikaza, kao što je ranije opisano.

Primjer 2-1. hello.py: Kompletan Flask aplikacija

```
iz flask import Flask app =
Flask(__name__)

@app.route('/') def
index(): return
'<h1>Hello World!</h1>'
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, sada možete pokrenuti git checkout 2a da provjerite ovu verziju aplikacije.

Web server za razvoj

Flask aplikacije uključuju razvojni web server koji se može pokrenuti naredbom flask run . Ova komanda traži ime Python skripte koja sadrži instancu aplikacije u varijabli okruženja FLASK_APP .

Da biste pokrenuli aplikaciju hello.py iz prethodnog odjeljka, prvo provjerite je li virtualno okruženje koje ste kreirali ranije aktivirano i da je u njemu instaliran Flask. Za korisnike Linuxa i macOS-a, pokrenite web server na sljedeći način:

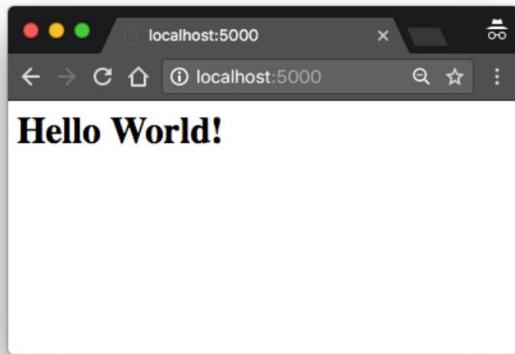
```
(venv) $ export FLASK_APP=hello.py (venv)
$ flask run * Serving Flask aplikacija
  "zdravo"
* Radi na http://127.0.0.1:5000/ (pritisnite CTRL+C za izlaz)
```

Za korisnike Microsoft Windows-a, jedina razlika je u tome kako je podešena varijabla okruženja FLASK_APP :

```
(venv) $ set FLASK_APP=hello.py (venv)
$ flask run * Serving Flask aplikacija
  "zdravo"
* Pokreće se na http://127.0.0.1:5000/ (pritisnite CTRL+C za izlaz)
```

Kada se server pokrene, ulazi u petlju koja prihvata zahtjeve i servisira ih. Ova petlja se nastavlja sve dok ne zaustavite aplikaciju pritiskom na Ctrl+C.

Dok je server pokrenut, otvorite svoj web pretraživač i upišite http://localhost:5000/ u adresnu traku. **Slika 2-1** pokazuje šta ćete videti nakon povezivanja na aplikaciju.



Slika 2-1. hello.py Flask aplikacija

Ako upišete bilo šta drugo nakon osnovnog URL-a, aplikacija neće znati kako se nositi s tim i vratit će šifru greške 404 u pretraživač – poznata greška koju dobijete kada se krećete do web stranice koja ne postoji.



Web server koji pruža Flask je namijenjen samo za razvoj i testiranje. Naučit ćete o proizvodnim web serverima u [17. poglavlju](#).



Flask razvojni web server se takođe može pokrenuti programski pozivanjem metode `app.run()`. Starije verzije Flaska koje nisu imale komandu `flask` zahtijevale su da se server pokrene pokretanjem glavne skripte aplikacije, koja je na kraju morala uključivati sljedeći isječak:

```
ako __name__ == '__main__':
    app.run()
```

Dok komanda `flask run` čini ovu praksu nepotrebnom, metoda `app.run()` i dalje može biti korisna u određenim prilikama, kao što je testiranje jedinica, kao što ćete naučiti u [poglavlju 15.](#)

Dynamic Routes

Druga verzija aplikacije, prikazana u [primjeru 2-2](#), dodaje drugu rutu koja je dinamička. Kada posjetite dinamički URL u vašem pretraživaču, prikazuje vam se personalizirani pozdrav koji uključuje ime navedeno u URL-u.

Primjer 2-2. hello.py: Flask aplikacija sa dinamičkom rutom

```
iz flask import Flask
app = Flask(__name__)

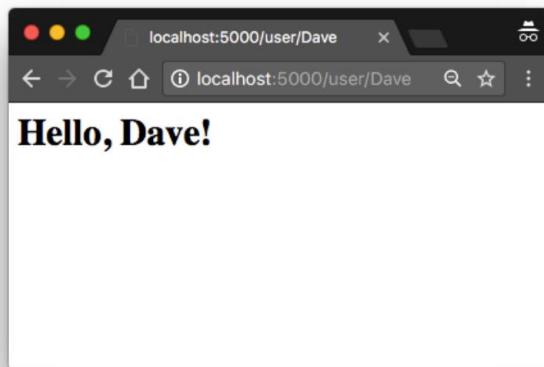
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Zdravo, {}!</h1>'.format(name)
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, sada možete pokrenuti git checkout 2b da provjerite ovu verziju aplikacije.

Da biste testirali dinamičku rutu, provjerite je li server pokrenut, a zatim idite na <http://localhost:5000/user/Dave>. Aplikacija će odgovoriti personaliziranim pozdravom koristeći dinamički argument imena. Pokušajte koristiti različita imena u URL-u da vidite kako funkcija prikaza uvijek generiše odgovor na osnovu datog imena. Primjer je prikazan na [slici 2-2](#).



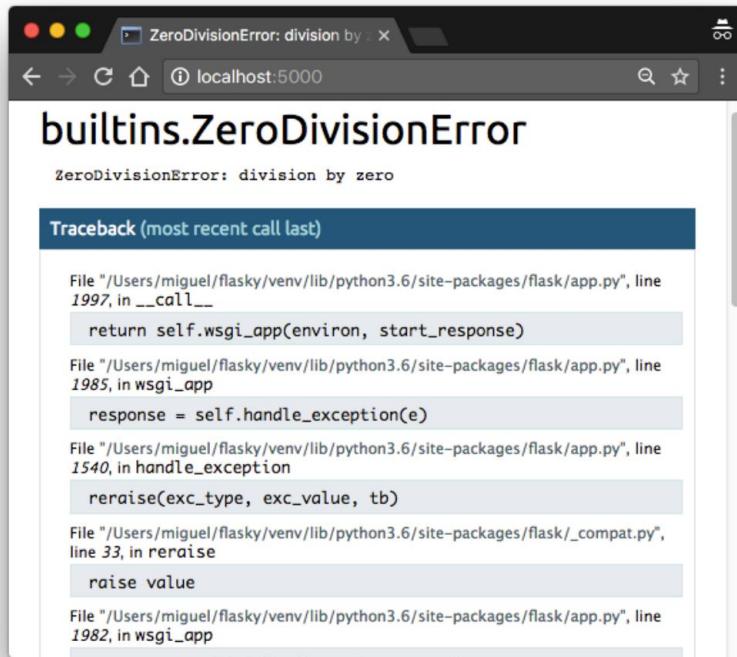
Slika 2-2. Dinamična ruta

Režim za otklanjanje grešaka

Flask aplikacije mogu opcionalno da se izvrše u debug modu. U ovom modu, dva vrlo zgodna modula razvojnog servera koji se nazivaju reloader i debugger su omogućena po defaultu.

Kada je ponovno učitavanje omogućeno, Flask prati sve datoteke izvornog koda vašeg projekta i automatski ponovo pokreće server kada se bilo koja od datoteka modifickira. Imati server koji radi sa uključenim reloaderom je izuzetno korisno tokom razvoja, jer svaki put kada modifikujete i sačuvate izvorni fajl, server se automatski ponovo pokreće i preuzima promene.

Debugger je alat zasnovan na webu koji se pojavljuje u vašem pretraživaču kada vaša aplikacija pokrene neobrađeni izuzetak. Prozor web pretraživača transformiše se u interaktivni stek koji vam omogućava da pregledate izvorni kod i procenite izraze na bilo kom mestu u steku poziva. Možete vidjeti kako debager izgleda na [slici 2-3](#).



Slika 2-3. Flask debugger

Podrazumevano, način za otklanjanje grešaka je onemogućen. Da biste to omogućili, postavite FLASK_DEBUG=1 varijablu okruženja prije pozivanja flask run:

```
(venv) $ export FLASK_APP=hello.py (venv) $ export
FLASK_DEBUG=1 (venv) $ flask run * Serving Flask
aplikacija "zdravo"

* Prisilno uključen način otklanjanja
grešaka * Pokreće se na http://127.0.0.1:5000/ (pritisnite CTRL+C da izadete)
* Ponovno pokretanje sa
statistikom * Debugger je aktiviran!
* PIN za otklanjanje grešaka: 273-181-528
```

Ako koristite Microsoft Windows, koristite set umjesto export za postavljanje varijabli okruženja.



Ako pokrenete svoj server metodom `app.run()`, variabile okruženja `FLASK_APP` i `FLASK_DEBUG` se ne koriste. Da biste programski omogućili način otklanjanja grešaka, koristite `app.run(debug=True)`.



Nikada ne omogućavajte način otklanjanja grešaka na proizvodnom serveru. Debager posebno omogućava klijentu da zatraži daljinsko izvršenje koda, tako da vaš proizvodni server čini ranjivim na napade. Kao jednostavna mjeru zaštite, program za otklanjanje grešaka mora biti aktiviran PIN-om, odštampanim na konzoli pomoću naredbe za pokretanje `flask`.

Opcije komandne linije

Komanda `flask` podržava brojne opcije. Da vidite šta je dostupno, možete pokrenuti `flask --help` ili jednostavno `flask` bez ikakvih argumenata:

```
(venv) $ flask --help Upotreba:  
flask [OPCIJE] KOMANDA [ARGS]...
```

Ova naredba ljudske djeluje kao opća uslužna skripta za Flask aplikacije.

Učitava konfiguriranu aplikaciju (preko variabile okruženja `FLASK_APP`), a zatim daje naredbe koje daje aplikacija ili sam Flask.

Najkorisnije komande su naredbe "run" i "shell".

Primjer upotrebe:

```
$ export FLASK_APP=hello.py $ export  
FLASK_DEBUG=1 $ flask run
```

Opcije: --
version Prikaži verziju flaske --help Prikaži ovu poruku i izađi.

naredbe:
trči Pokreće razvojni server. shell Pokreće ljudsku u kontekstu aplikacije.

Naredba `flask shell` se koristi za pokretanje Python sesije ljudske u kontekstu aplikacije. Ovu sesiju možete koristiti za pokretanje zadataka održavanja ili testova ili za otklanjanje grešaka. Stvarni primjeri u kojima je ova naredba korisna bit će predstavljeni kasnije, u nekoliko poglavlja.

Već ste upoznati sa naredbom flask run , koja, kao što joj naziv govori, pokreće aplikaciju sa razvojnim web serverom. Ova komanda ima mnogo opcija:

```
(venv) $ flask run --help
Upotreba: trčanje u boci [OPCIJE]
```

Pokreće lokalni razvojni server za Flask aplikaciju.

Ovaj lokalni server se preporučuje samo za razvojne svrhe, ali se može koristiti i za jednostavnu intranet implementaciju. Prema zadanim postavkama, neće podržavati bilo kakvu paralelnost da bi se pojednostavilo otklanjanje grešaka. Ovo se može promijeniti opcijom --with-threads koja će omogućiti osnovno multithreading.

Ponovno učitavanje i program za otklanjanje grešaka su podrazumevano omogućeni ako je debug flag Flask omogućen ili onemogućen u suprotnom.

Opcije: -h, --

host TEKST -p, --port	Interfejs za povezivanje.
INTEGER --reload / --no-	Port za povezivanje.
reload	Omogućite ili onemogućite ponovno učitavanje. Po defaultu je reloader aktivan ako je debug omogućen.
--debugger / --no-debugger	Omogućite ili onemogućite program za otklanjanje grešaka. Podrazumevano je debugger aktivan ako je debug omogućen.
--eager-loading / --lazy-loader	Omogućite ili onemogućite željno učitavanje. Podrazumevano je željno učitavanje omogućeno ako je reloader onemogućen. --with-threads / --without-threads
-pomoć	Omogućite ili onemogućite višenitost. Prikažite ovu poruku i izadite.

Argument --host je posebno koristan jer govori web serveru koji mrežni interfejs treba da sluša za konekcije od klijenata. Podrazumevano, Flaskov razvojni veb server osluškuje veze na lokalnom hostu, tako da se prihvataju samo veze koje potiču sa računara koji pokreće server. Sljedeća naredba čini da web server osluškuje veze na javnom mrežnom interfejsu, omogućavajući i drugim računarima u istoj mreži da se povežu:

```
(venv) $ flask run --host 0.0.0.0 * Serving Flask
aplikacija "zdravo"
* Pokreće se na http://0.0.0.0:5000/ (pritisnite CTRL+C da izadete)
```

Web server bi sada trebao biti dostupan sa bilo kog računara u mreži na adresi http:// abcd:5000, gdje je abcd IP adresa računara koji pokreće server u vašoj mreži.

Opcije --reload, --no-reload, --debugger i --no-debugger pružaju veći stepen kontrole na vrhu postavke načina otklanjanja grešaka. Na primjer, ako debug

mod je omogućen, --no-debugger se može koristiti za isključivanje programa za otklanjanje grešaka, uz zadržavanje režima za otklanjanje grešaka i ponovnog učitavanja uključenih.

Ciklus zahtjev-odgovor

Sada kada ste se igrali sa osnovnom aplikacijom Flask, možda biste željeli znati više o tome kako Flask radi svoju magiju. Sljedeći odjeljci opisuju neke od aspekata dizajna okvira.

Konteksti aplikacije i zahtjeva Kada Flask

primi zahtjev od klijenta, mora nekoliko objekata učiniti dostupnim funkciji prikaza koja će njime upravljati. Dobar primjer je objekt zahtjeva, koji inkapsulira HTTP zahtjev koji šalje klijent.

Očigledan način na koji bi Flask mogao dati funkciji pogleda pristup objektu zahtjeva je slanje istog kao argumenta, ali to bi zahtjevalo da svaka pojedinačna funkcija pogleda u aplikaciji ima dodatni argument. Stvari postaju složenije ako uzmete u obzir da objekt zahtjeva nije jedini objekt kojem funkcije pregleda možda trebaju pristupiti da bi ispunile zahtjev.

Da bi se izbjeglo zatrpanjanje funkcija pogleda s puno argumenata koji možda nisu uvijek potrebni, Flask koristi kontekste da privremeno učini određene objekte globalno dostupnim. Zahvaljujući kontekstima, funkcije prikaza kao što je ova mogu se napisati:

`iz zahtjeva za uvoz tikvice`

```
@app.route('/')
def
index():
    user_agent = request.headers.get('User-Agent') return
    '<p>Vaš pregleđnik je {}</p>'.format(user_agent)
```

Obratite pažnju na to kako se u ovoj funkciji pogleda zahtjev koristi kao da je globalna varijabla. U stvarnosti, zahtjev ne može biti globalna varijabla; u višenitnom serveru nekoliko niti može raditi na različitim zahtjevima od različitih klijenata sve u isto vrijeme, tako da svaka nit mora vidjeti drugačiji objekt u zahtjevu. Konteksti omogućavaju Flasku da određene varijable učini globalno dostupnim niti bez ometanja drugih niti.



Nit je najmanji niz instrukcija kojim se može upravljati nezavisno. Uobičajeno je da proces ima više aktivnih niti, ponekad dijeleći resurse kao što su memorija ili ručke datoteka. Višenitni web serveri pokreću skup niti i biraju nit iz skupa za rukovanje svakim dolaznim zahtjevom.

U Flasku postoje dva konteksta: kontekst aplikacije i kontekst zahtjeva.

Tabela 2-1 prikazuje varijable izložene u svakom od ovih konteksta.

Tabela 2-1. Globalni kontekst flask

Ime varijable Kontekst		Opis Instanca
current_app	Kontekst aplikacije	aplikacije za aktivnu aplikaciju.
g	Aplikacija kontekstu	Objekt koji aplikacija može koristiti za privremenu memoriju za vrijeme obrade zahtjeva. Ova varijabla se resetuje sa svakim zahtjevom.
zahtjev	Kontekst zahtjeva	Objekt zahtjeva, koji sadrži sadržaj HTTP zahtjeva koji je poslao klijent.
sjednici	Kontekst zahtjeva	Korisnička sesija, rječnik koji aplikacija može koristiti za pohranjivanje vrijednosti koje se "pamte" između zahtjeva.

Flask aktivira (ili gura) kontekst aplikacije i zahtjeva prije slanja zahtjeva aplikaciji i uklanja ih nakon što se zahtjev obradi. Kada je kontekst aplikacije gurnut, varijable current_app i g postaju dostupne niti. Isto tako, kada je kontekst zahtjeva gurnut, zahtjev i sesija također postaju dostupni. Ako se bilo kojoj od ovih varijabli pristupi bez aktivne aplikacije ili konteksta zahtjeva, generira se greška. Četiri kontekstualne varijable će biti detaljno obrađene u ovom i kasnjim poglavljima, tako da ne brinite ako još ne razumijete zašto su korisne.

Sljedeća sesija Python ljske pokazuje kako funkcioniра kontekst aplikacije:

```
>>> iz aplikacije hello import >>>
iz flask import current_app >>>
current_app.name Traceback (zadnji posljednji
poziv):
...
RuntimeError: radi izvan konteksta aplikacije >>> app_ctx =
app.app_context() >>> app_ctx.push() >>> current_app.name 'zdravo'
>>> app_ctx.pop()
```

U ovom primjeru, current_app.name ne uspijeva kada nije aktivan kontekst aplikacije, ali postaje važeći nakon što se kontekst aplikacije za aplikaciju gurne. Obratite pažnju na to kako se kontekst aplikacije dobija pozivanjem app.app_context() na instanci aplikacije.

Otpremanje zahtjeva Kada

aplikacija primi zahtjev od klijenta, mora saznati koju funkciju pregleda da pozove da bi je servisirala. Za ovaj zadatak, Flask traži URL dat u

zahtjev u URL mapi aplikacije, koja sadrži mapiranje URL-ova u funkcije prikaza koje njima rukuju. Flask gradi ovu mapu koristeći podatke date u dekoratoru `app.route`, ili ekvivalentnoj verziji bez dekoratora, `app.add_url_rule()`.

Da biste vidjeli kako izgleda URL mapa u aplikaciji Flask, možete pregledati mapu kreiranu za `hello.py` u Python ljudi. Prije nego što ovo pokušate, uvjerite se da je vaše virtuelno okruženje aktivirano:

```
(venv) $ python >>> iz
aplikacije hello import >>> app.url_map
Map([<Pravilo '/' (HEAD, OPTIONS, GET)
-> index>, <Pravilo '/static/<filename>' (HEAD , OPCIJE, GET) -> statički>,
<Pravilo '/user/<ime>' (GLAVA, OPCIJE, GET) -> korisnik>])
```

Rute i `/user/<name>` definirali su dekoratori `app.route` u aplikaciji. `/static/<filename>` ruta je posebna ruta koju je dodao Flask da bi se omogućio pristup statičnim datotekama. Više o statičkim datotekama čete naučiti u 3. poglavlju .

Elementi (HEAD, OPTIONS, GET) prikazani u URL mapi su metode zahtjeva kojima rute rukuju. HTTP specifikacija definira da se svi zahtjevi izdaju s metodom, koja obično pokazuje koju radnju klijent traži od servera da izvrši. Flask prilaže metode svakoj ruti tako da različite metode zahtjeva poslane na isti URL mogu biti rukovane različitim funkcijama prikaza. Flask automatski upravlja metodama HEAD i OPTIONS , tako da se u praksi može reći da su u ovoj aplikaciji tri rute u URL mapi pridružene GET metodi, koja se koristi kada klijent želi zatražiti informacije kao što je web stranica. Naučit ćete kako kreirati rute za druge metode zahtjeva u 4. poglavlju .

Objekt zahtjeva Vidjeli

ste da Flask izlaže objekt zahtjeva kao kontekstualnu varijablu pod nazivom zahtjev. Ovo je izuzetno koristan objekat koji sadrži sve informacije koje je klijent uključio u HTTP zahtjev. **Tabela 2-2** nabroja najčešće korištene atribute i metode objekta zahtjeva Flask.

Tabela 2-2. Objekat zahtjeva za flask

Opis atributa ili metode	
formu	Rječnik sa svim poljima obrasca dostavljenim uz zahtjev.
args	Rječnik sa svim argumentima proslijeđenim u nizu upita URL-a.
vrijednosti	Rječnik koji kombinira vrijednosti u formi i argumentima.
kolačići	Rječnik sa svim kolačićima uključenim u zahtjev.
zaglavljiva	Rječnik sa svim HTTP zaglavljima uključenim u zahtjev.
datoteke	Rječnik sa svim učitanim datotekama uključenim u zahtjev.

Atribut ili Metod Opis Vraća

<code>get_data()</code>	podatke u baferu iz tijela zahtjeva.
<code>get_json()</code>	Vraća Python rječnik s raščlanjenim JSON-om uključenim u tijelo zahtjeva.
<code>nacrt</code>	Ime Flask nacrt koji obrađuje zahtjev. Nacrti su predstavljeni u Poglavlju 7 .
<code>krajnja tačka</code>	Ime Flask krajnje tačke koja obrađuje zahtjev. Flask koristi ime funkcije prikaza kao naziv krajnje točke za rutu.
<code>metoda</code>	Metoda HTTP zahtjeva, kao što je GET ili POST.
<code>shema</code>	URL šema (http ili https).
<code>je_sigurno()</code>	Vraća True ako je zahtjev došao preko sigurne (HTTPS) veze.
<code>domaćin</code>	Host definiran u zahtjevu, uključujući broj porta ako ga je dao klijent.
<code>put</code>	Dio putanje URL-a.
<code>query_string</code>	Dio niza upita URL-a, kao sirova binarna vrijednost.
<code>full_path</code>	Dijelovi putanje i stringa upita URL-a.
<code>url</code>	Potpuni URL koji je zatražio klijent.
<code>base_url</code>	Isto kao i url, ali bez komponente stringa upita.
<code>remote_addr</code>	IP adresa klijenta.
<code>o</code>	Neobrađeni rječnik WSGI okruženja za zahtjev.

Zakačivanje

zahtjeva Ponekad je korisno izvršiti kod prije ili nakon obrade svakog zahtjeva. Na primjer, na početku svakog zahtjeva može biti potrebno kreirati vezu sa bazom podataka ili autentifikovati korisnika koji postavlja zahtjev. Umjesto dupliranja koda koji izvodi ove radnje u svakoj funkciji pogleda, Flask vam daje opciju da registrijete uobičajene funkcije koje će se pozivati prije ili nakon slanja zahtjeva.

Kuke za zahtjeve su implementirane kao dekoratori. Ovo su četiri kuke koje Flask podržava:

`before_request`

Registrira funkciju za pokretanje prije svakog zahtjeva.

`before_first_request`

Registrira funkciju za pokretanje samo prije nego što se obradi prvi zahtjev. Ovo može biti zgodan način za dodavanje zadataka inicijalizacije servera.

`after_request`

Registrira funkciju za pokretanje nakon svakog zahtjeva, ali samo ako nije došlo do neobrađenih izuzetaka.

`teardown_request`

Registrira funkciju za pokretanje nakon svakog zahtjeva, čak i ako su se desili neobrađeni izuzeci.

Uobičajeni obrazac za dijeljenje podataka između funkcija zakačivanja zahtjeva i funkcija pregleda je korištenje globalnog konteksta g kao pohrane. Na primjer, obrađivač before_request može učitati prijavljenog korisnika iz baze podataka i pohraniti ga u g.user. Kasnije, kada se pozove funkcija pregleda, ona može preuzeti korisnika odatle.

U narednim poglavljima će biti prikazani primjeri zakačiva za zahtjeve, tako da ne brinite ako svrha ovih kukica još uvijek nema smisla.

Odgovori

Kada Flask pozove funkciju pogleda, očekuje da njena povratna vrijednost bude odgovor na zahtjev. U većini slučajeva odgovor je jednostavan niz koji se šalje nazad klijentu kao HTML stranica.

Ali HTTP protokol zahtijeva više od niza kao odgovor na zahtjev. Vrlo važan dio HTTP odgovora je statusni kod, koji Flask po defaultu postavlja na 200, kod koji označava da je zahtjev uspješno obavljen.

Kada funkcija pogleda treba da odgovori drugim statusnim kodom, može dodati numerički kod kao drugu povratnu vrijednost nakon teksta odgovora. Na primjer, sljedeća funkcija pregleda vraća statusni kod 400, kod za grešku lošeg zahtjeva:

```
@app.route('/')
def index():
    vraća
    '<h1>Loš zahtjev</h1>', 400
```

Odgovori koje vraćaju funkcije pregleda mogu također uzeti treći argument, rječnik zaglavlj koja se dodaju u HTTP odgovor. Vidjet ćete primjer prilagođenih zaglavja odgovora u [14. poglavlju](#).

Umjesto vraćanja jedne, dvije ili tri vrijednosti kao tuple, funkcije Flask view imaju opciju vraćanja objekta odgovora. Funkcija make_response() uzima jedan, dva ili tri argumenta, iste vrijednosti koje se mogu vratiti iz funkcije pogleda, i vraća ekvivalentni objekt odgovora. Ponekad je korisno generirati objekt odgovora unutar funkcije prikaza, a zatim koristiti njegove metode za dalje konfiguriranje odgovora. Sljedeći primjer kreira objekt odgovora i zatim postavlja kolačić u njega:

```
iz flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>Ovaj dokument nosi kolačić!</h1>')
    response.set_cookie('answer', '42')
    vrati odgovor
```

[Tabela 2-3](#) prikazuje najčešće korištene atribute i metode dostupne u objektima odgovora.

Tabela 2-3. Objekt odgovora tikvice

Opis atributa ili metode	
status_code	Numerički HTTP statusni kod
zaglavljia	Objekt sličan rječniku sa svim zaglavljima koji će biti poslati s odgovorom
set_cookie()	Dodaje kolačić odgovoru
delete_cookie()	Uklanja kolačić
content_length	Dužina tijela odgovora
content_type	Tip medija tijela odgovora
set_data()	Postavlja tijelo odgovora kao vrijednost niza ili bajtova
get_data()	Dobiva tijelo odgovora

Postoji posebna vrsta odgovora koja se zove preusmjeravanje. Ovaj odgovor ne uključuje dokument stranice; samo daje pretraživaču novi URL do kojeg može navigirati. Vrlo uobičajena upotreba preusmjeravanja je kada radite s web obrascima, kao što ćete naučiti u 4. poglavlju .

Preusmjeravanje je obično označeno statusnim kodom odgovora 302 i URL-om na koji se ide datim u zaglavljtu lokacije . Odgovor za preusmjeravanje može se generirati ručno s povratom od tri vrijednosti ili sa objektom odgovora, ali s obzirom na njegovu čestu upotrebu, Flask pruža pomoćnu funkciju redirect() koja kreira ovaj tip odgovora:

```
iz preusmjeravanja uvoza flask

@app.route('/') def
index():
    return redirect('http://www.example.com')
```

Drugi poseban odgovor se izdaje sa funkcijom abort() , koja se koristi za rukovanje greškama. Sljedeći primjer vraća statusni kod 404 ako dinamički argument id dat u URL-u ne predstavlja važećeg korisnika:

```
iz flask import prekida

@app.route('/user/<id>') def
get_user(id): user = load_user(id) ako
nije user: abort(404) return
'<h1>Zdravo, {}</h1>'.format
(user.name)
```

Imajte na umu da abort() ne vraća kontrolu natrag u funkciju jer pokreće izuzetak.

Flask Extensions

Boca je dizajnirana da se produži. Namjerno se zadržava izvan područja važne funkcionalnosti, kao što su baza podataka i autentikacija korisnika, dajući vam slobodu da odaberete pakete koji najbolje odgovaraju vašoj aplikaciji ili da napišete svoje, ako to želite.

Zajednica je kreirala veliki izbor Flask ekstenzija za mnoge različite svrhe, a ako to nije dovoljno, može se koristiti i bilo koji standardni Python paket ili biblioteka. Svoj prvi Flask ekstenziju čete koristiti u 3. poglavlju .

Ovo poglavlje uvodi koncept odgovora na zahtjeve, ali ima još puno toga za reći o odgovorima. Flask pruža veoma dobru podršku za generisanje odgovora pomoću šablonu, a ovo je toliko važna tema da je sledeće poglavlje posvećeno njoj.

Predlošci

Ključ za pisanje aplikacija koje se lako održavaju je pisanje čistog i dobro strukturiranog koda. Primeri koje ste do sada videli su suviše jednostavnici da bi to demonstrirali, ali funkcije Flask view imaju dve potpuno nezavisne svrhe prerušene u jednu, što stvara problem.

Očigledan zadatak funkcije prikaza je generiranje odgovora na zahtjev, kao što ste vidjeli u primjerima prikazanim u [Poglavlju 2](#). Za najjednostavnije zahtjeve to je dovoljno, ali u mnogim slučajevima zahtjev također pokreće promjenu stanja aplikacija, a funkcija prikaza je mjesto gdje se ova promjena generiše.

Na primjer, uzmite u obzir korisnika koji registruje novi nalog na web stranici. Korisnik upisuje adresu e-pošte i lozinku u web obrazac i klikne na dugme Pošalji.

Na serveru stiže zahtjev sa podacima koje je korisnik dostavio, a Flask ga šalje funkciji pregleda koja obrađuje zahtjeve za registraciju. Ova funkcija pregleda mora razgovarati s bazom podataka kako bi dodala novog korisnika, a zatim generirala odgovor za slanje natrag pregledniku koji uključuje poruku o uspjehu ili neuspjehu. Ove dvije vrste zadataka se formalno nazivaju poslovna logika i logika prezentacije, respektivno.

Miješanje poslovne i prezentacijske logike dovodi do koda koji je teško razumjeti i održavati. Zamislite da morate da napravite HTML kod za veliku tabelu spajanjem podataka dobijenih iz baze podataka sa potrebnim literalima HTML stringova. Premještanje logike prezentacije u šablone pomaže poboljšanju mogućnosti održavanja aplikacije.

Šablon je datoteka koja sadrži tekst odgovora, sa varijablama za čuvanje mjesta za dinamičke dijelove koji će biti poznati samo u kontekstu zahtjeva. Proces koji zamjenjuje varijable stvarnim vrijednostima i vraća konačni niz odgovora naziva se renderiranje. Za zadatak renderiranja predložaka, Flask koristi moćni mehanizam za šablove koji se zove Jinja2.

Jinja2 Template Engine

U svom najjednostavnijem obliku, Jinja2 šablon je datoteka koja sadrži tekst odgovora. **Primjer 3-1** prikazuje predložak Jinja2 koji odgovara odgovoru funkcije prikaza index() iz primjera 2-1.

Primjer 3-1. templates/index.html: Jinja2 šablon

```
<h1>Zdravo svijetel</h1>
```

Odgovor koji vraća funkcija pogleda user() iz **primjera 2-2** ima dinamičku komponentu, koja je predstavljena promjenljivom. **Primjer 3-2** pokazuje predložak koji implementira ovaj odgovor.

Primjer 3-2. templates/user.html: šablon Jinja2

```
<h1>Zdravo, {{ name }}!</h1>
```

Rendering Templates

Standardno Flask traži šablone u poddirektoriju šablonu koji se nalazi unutar glavnog direktorijuma aplikacije. Za sljedeću verziju hello.py, potrebno je da kreirate poddirektorijum šablonu i da u njega pohranite šablone definisane u prethodnim primerima kao index.html i user.html.

Funkcije prikaza u aplikaciji moraju biti modificirane da bi se ovi predlošci prikazali. **Primjer 3-3** pokazuje ove promjene.

Primjer 3-3. hello.py: prikazivanje šablonu

```
iz flask import Flask, render_template

# ...

@app.route('/') def
index(): return
    render_template('index.html')

@app.route('/user/<ime>') def
user(name): return
    render_template('user.html', name=name)
```

Funkcija `render_template()` koju pruža Flask integriše Jinja2 šablonski mehanizam sa aplikacijom. Ova funkcija uzima ime datoteke predloška kao prvi argument. Svi dodatni argumenti su parovi ključ/vrijednost koji predstavljaju stvarne vrijednosti za varijable na koje se upućuje u predlošku. U ovom primjeru, drugi predložak prima varijablu imena .

Argumenti ključnih riječi kao što je `ime=ime` u prethodnom primjeru su prilično uobičajeni, ali mogu izgledati zbumnjuće i teško razumjeti ako niste navikli na njih. „Ime“ na levoj strani predstavlja ime argumenta, koje se koristi u čuvaru mesta napisanom u šablonu. „Ime“ na desnoj strani je varijabla u trenutnom opsegu koja daje vrijednost za argument istog imena. Iako je ovo uobičajen obrazac, korištenje istog imena varijable na obje strane nije potrebno.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti `git checkout 3a` da provjerite ovu verziju aplikacije.

Varijable

Konstrukcija `{{ name }}` koja se koristi u predlošku prikazanom u [primjeru 3-2](#) upućuje na varijablu, posebno mjesto za čuvanje mjesta koje govori mehanizmu šablona da vrijednost koja ide na to mjesto treba biti dobijena iz podataka koji su dati u trenutku kada je šablon ren - dered.

Jinja2 prepoznaće varijable bilo kojeg tipa, čak i složene tipove kao što su liste, rječnici i objekti. Slijedi još nekoliko primjera varijabli koje se koriste u predlošcima:

```
<p> Vrijednost iz rječnika: {{ mydict['key'] }}.</p> <p> Vrijednost iz liste: {{ mylist[3] }}.</p> <p> A vrijednost sa liste, sa varijabilnim indeksom: {{ mylist[myintvar] }}.</p> <p> Vrijednost iz metode objekta: {{ myobj.somemethod() }}.</p>
```

Varijable se mogu modifikovati pomoću Itera, koji se dodaju iza imena varijable sa znakom za crtu kao separatorom. Na primjer, sljedeći predložak prikazuje promjenljivu imena velikim slovom:

```
Zdravo, {{ ime|velika_slova }}
```

[Tabela 3-1](#) navodi neke od najčešće korištenih filtera koji dolaze s Jinja2.

Tabela 3-1. Jinja2 varijabilni litri

Naziv filtera	Opis
sigurno	Renderira vrijednost bez primjene izbjegavanja
kapitalize	Pretvara prvi znak vrijednosti u velika slova, a ostatak u mala slova
niže	Pretvara vrijednost u mala slova
gornji	Pretvara vrijednost u velika slova
naslov	Svaku riječ u vrijednosti piše velikim slovom
podrezati	Uklanja početne i zadnje razmake iz traka vrijednosti
	Uklanja sve HTML oznake iz vrijednosti prije renderiranja

Zanimljivo je istaknuti siguran filter. Po defaultu Jinja2 izbjegava sve varijable iz sigurnosnih razloga. Na primjer, ako je varijabla postavljena na vrijednost '<h1>Zdravo</h1>', Jinja2 će string prikazati kao '<h1>Hello</h1>', što će uzrokovati da se element h1 prikaže i ne tumači pretraživač. Mnogo puta je potrebno prikazati HTML kod pohranjen u varijablama, a za te slučajeve se koristi siguran filter.



Nikada nemojte koristiti siguran filter za vrijednosti koje nisu pouzdane, kao što je tekst koji su korisnici unijeli u web obrasce.

Kompletan listu filtera možete dobiti iz službene [Jinja2 dokumentacije](#).

Kontrolne strukture

Jinja2 nudi nekoliko kontrolnih struktura koje se mogu koristiti za promjenu toka šablona. Ovaj odjeljak uvodi neke od najkorisnijih s jednostavnim primjerima.

Sljedeći primjer pokazuje kako se uvjetni iskazi mogu unijeti u predložak:

```
{% ako korisnik %}
    Zdravo, {{ korisnik }}!
{% ostalo %}
    Zdravo stranče! {%
    endif %}
```

Još jedna uobičajena potreba u šablonima je da se prikaže lista elemenata. Ovaj primjer pokazuje kako se to može učiniti sa for petljom:

```
<ul>
    {% za komentar u komentarima %}
```

```
<li>{{ comment }}</li> {%
endfor %} </ul>
```

Jinja2 također podržava makronaredbe, koje su slične funkcijama u Python kodu. Na primjer:

```
{% macro render_comment(comment) %}
<li>{{ comment }}</li> {% endmacro %}

<ul>
{%- za komentar u komentarima %}
    {{ render_comment(comment) }} {%
endfor %} </ul>
```

Da bi makroi bili višekratni upotrebljivi, mogu se pohraniti u samostalne datoteke koje se zatim uvoze iz svih predložaka kojima su potrebni:

```
{% import 'macros.html' kao makronaredbe
%} <ul> {%- za komentar u komentarima %}
    {{ macros.render_comment(comment) }}

    {% endfor %}
</ul>
```

Dijelovi koda šablona koji se moraju ponoviti na nekoliko mesta mogu se pohraniti u zasebnu datoteku i uključiti iz svih predložaka kako bi se izbjeglo ponavljanje:

```
{% uključuje 'common.html' %}
```

Još jedan moćan način ponovne upotrebe je nasljeđivanje šablona, koje je slično nasljeđivanju klase u Python kodu. Prvo se kreira osnovni predložak s imenom base.html:

```
<html>
<glava>
    {% block head %}
        <title>{% block title %}{% endblock %} - Moja aplikacija</title> {% endblock %}
    </head> <body> {% block body %} {% endblock %} </body> </html>
```

Osnovni predlošci definiraju blokove koji se mogu nadjačati izvedenim predlošcima. Direktiva Jinja2 block i endblock definiraju blokove sadržaja koji se dodaju bazi šablon. U ovom primjeru postoje blokovi koji se zovu glava, naslov i tijelo; imajte na umu da je naslov sadržan u zaglavlju. Sljedeći primjer je izvedeni predložak osnovnog predloška:

```
{% proširuje "base.html" %} {%
block title %}Indeks{% endblock %} {% glava
bloka %} {{ super() }} <style> </style> {% 
endblock %} {% blok tijelo %}
<h1>Zdravo, svijetel</h1> {% endblock
%}
```

Direktiva extends izjavljuje da je ovaj predložak izведен iz base.html. Nakon ove direktive slijede nove definicije za tri bloka definirana u osnovnom predlošku, koji su umetnuti na odgovarajuća mesta. Kada blok ima neki sadržaj iu osnovnom iu izvedenom predlošku, koristi se sadržaj iz izvedenog predloška. Unutar ovog bloka, izvedeni predložak može pozvati super() da referencira sadržaj bloka u osnovnom predlošku. U prethodnom primjeru, to se radi u bloku glave .

Stvarna upotreba svih kontrolnih struktura predstavljenih u ovom odeljku biće prikazana kasnije, tako da čete imati priliku da vidite kako one rade.

Bootstrap integracija sa Flask-Bootstrapom

Bootstrap je okvir web pretraživača otvorenog koda iz Twittera koji pruža komponente korisničkog sučelja koje pomažu u kreiranju čistih i atraktivnih web stranica koje su kompatibilne sa svim modernim web pretraživačima koji se koriste na desktop i mobilnim platformama.

Bootstrap je okvir na strani klijenta, tako da server nije direktno uključen u njega. Sve što server treba da uradi jeste da obezbedi HTML odgovore koji upućuju na Bootstrapove kaskadne listove stilova (CSS) i JavaScript datoteke, i instanciraju željene elemente korisničkog interfejsa kroz HTML, CSS i JavaScript kod. Idealno mjesto za sve ovo je u šablonima.

Naivan pristup integraciji Bootstrapa sa aplikacijom je da izvršite sve potrebne promjene u HTML šablonima, slijedeći preporuke date u Bootstrap dokumentaciji. Ali ovo je oblast u kojoj upotreba ekstenzije Flask čini zadatku integracije mnogo jednostavnijim, dok istovremeno pomaže da ove promene budu lepo organizovane.

Ekstenzija se zove Flask-Bootstrap i može se instalirati pomoću pip-a:

```
(venv) $ pip install flask-bootstrap
```

Ekstenzije flask-a se inicijaliziraju u isto vrijeme kada se kreira instanca aplikacije.

Primer 3-4 pokazuje inicijalizaciju Flask-Bootstrapa.

Primjer 3-4. hello.py: Flask-Bootstrap inicijalizacija

```
from flask_bootstrap import Bootstrap #
bootstrap = Bootstrap(app)
```

Ekstenzija se obično uvozi iz flask_<ime> paketa, gdje je <ime> naziv ekstenzije. Većina ekstenzija Flask prati jedan od dva konzistentna obrasca za inicijalizaciju. U **primjeru 3-4**, ekstenzija se inicijalizira proslijedivanjem instance aplikacije kao argumenta u konstruktoru. Naučit ćete o naprednijoj metodi za inicijalizaciju ekstenzija prikladnih za veće aplikacije u 7. poglavlju .

Kada se Flask-Bootstrap inicijalizira, osnovni predložak koji uključuje sve Bootstrap datoteke i opću strukturu je dostupan aplikaciji. Aplikacija zatim koristi prednosti nasleđivanja predloška Jinja2 da proširi ovaj osnovni predložak. **Primjer 3-5** prikazuje novu verziju user.html kao izvedeni predložak.

Primjer 3-5. templates/user.html: šablon koji koristi Flask-Bootstrap

```
{% proširuje "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation"> <div
class="container"> <div class="navbar-header">

    <button type="button" class="navbar-toggle" data-
    toggle="collapse" data-target=".navbar-collapse"> <span class="sr-
    only">Uključi navigaciju <span class="icon-bar"> <span
class="icon-bar"> <span class="icon-bar"> </button> <a
class="navbar-brand" href="/">Flasky</a> </div> <div
class="navbar-collapse collapse">

        <ul class="none navbar-none">
            <li><a href="/">Početna</a></li> </
        ul> </div> </div> </div> {% endblock %}

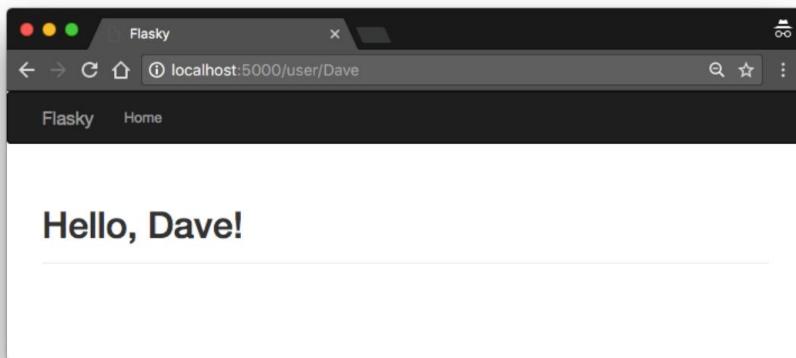
{% block content %}
<div class="container"> <div
class="page-header">
    <h1>Zdravo, {{ name }}!</h1> </
    div>
```

```
</div>
{% endblock %}
```

Direktiva Jinja2 extends implementira nasljeđivanje šablona upućivanjem na bootstrap/base.html iz Flask-Bootstrapa. Osnovni predložak iz Flask-Bootstrapa pruža skelet web stranice koja uključuje sve Bootstrap CSS i JavaScript datoteke.

Šablon user.html definira tri bloka pod nazivom naslov, navigacijska traka i sadržaj. Ovo su svi blokovi koje osnovni predložak izvozi za definirane izvedene predloške. Naslovni blok je jednostavan; njegov sadržaj će se pojaviti između oznaka <title> u zaglavlje prikazanog HTML dokumenta. Navbar i blokovi sadržaja rezervirani su za traku za navigaciju stranica i glavni sadržaj.

U ovom predlošku, blok navigacijske trake definira jednostavnu navigacijsku traku koristeći Bootstrap komponente. Blok sadržaja ima kontejner <div> sa zaglavljem stranice unutra. Pozdravna linija koja je bila u prethodnoj verziji šablona sada je unutar zaglavlja stranice. [Slika 3-1](#) pokazuje kako aplikacija izgleda sa ovim promjenama.



Slika 3-1. Bootstrap šabloni



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 3b da provjerite ovu verziju aplikacije. Flask-Bootstrap paket takođe treba da bude instaliran u vašem virtuelnom okruženju. [Službena dokumentacija](#) Bootstrapa je odličan resurs za učenje pun primjera spremnih za kopiranje/lijepljenje.

Flask-Bootstrap-ov base.html šablon definira nekoliko drugih blokova koji se mogu koristiti u izvedenim predlošcima. [Tabela 3-2](#) prikazuje kompletну listu dostupnih blokova.

Tabela 3-2. Flask-Bootstrap-ov osnovni predložak blokovi

Naziv bloka	Opis
doc	Cijeli HTML dokument
html_attribs	Atributi unutar <html> oznake
html	Sadržaj oznake <html>
glava	Sadržaj oznake <head>
naslov	Sadržaj oznake <title>
ciljevi	Lista <meta> oznaka
stilova	CSS definicije
body_attribs	Atributi unutar tijela oznake <body>
	Sadržaj oznake <body>
navbar	Korisnički definirana navigacijska traka
sadržaj	Korisnički definiran sadržaj stranice
skripte	JavaScript deklaracije na dnu dokumenta

Mnoge blokove u **Tabeli 3-2** koristi sam Flask-Bootstrap, tako da bi njihovo direktno nadjačavanje izazvalo probleme. Na primjer, blokovi stilova i skripti su mjesto gdje se deklariraju Bootstrap CSS i JavaScript datoteke. Ako aplikacija treba da doda svoj sadržaj u blok koji već ima neki sadržaj, tada se mora koristiti funkcija `Jinja2 super()`. Na primjer, ovako bi blok skripti trebao biti napisan u izvedenom predlošku da bi se dodala nova JavaScript datoteka u dokument:

```
{% blok skripti %} {{ super() }}
<script type="text/javascript"
src="my-script.js"></script> {% endblock %}
```

Stranice prilagođenih grešaka

Kada unesete nevažeću rutu u adresnu traku vašeg pretraživača, dobićete stranicu greške koda 404. U poređenju sa stranicama koje pokreće Bootstrap, zadana stranica o grešci je sada previše obična i neprivlačna i nema konzistentnost sa stvarnim stranicama koje je generirala aplikacija.

Flask omogućava aplikaciji da definiše prilagođene stranice grešaka koje se mogu zasnovati na šablonima, poput redovnih ruta. Dvije najčešće šifre greške su 404, koji se pokreće kada klijent zatraži stranicu ili rutu koja nije poznata, i 500, koja se pokreće kada postoji neobrađeni izuzetak u aplikaciji. **Primer 3-6** pokazuje kako obezbediti prilagođene rukovaće za ove dve greške pomoću dekoratora `app.errorhandler`.

Primjer 3-6. hello.py: prilagođene stranice grešaka

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e): return
    render_template('500.html'), 500
```

Rukovaoci greškama vraćaju odgovor, poput funkcija pregleda, ali takođe moraju da vrate numerički statusni kod koji odgovara grešci, što Flask prikladno prihvata kao drugu povratnu vrednost.

Predlošci na koje se upućuje u obrađivačima grešaka moraju biti napisani. Ovi predlošci bi trebali imati isti izgled kao i obične stranice, tako da će u ovom slučaju imati traku za navigaciju i zaglavje stranice koje prikazuje poruku o grešci.

Jednostavan način za pisanje ovih predložaka je da kopirate templates/user.html u templates/404.html i templates/500.html i zatim promijenite elemente zaglavlja stranice u ove dvije nove datoteke u odgovarajuće poruke o grešci, ali to će generirati puno duplikiranja.

Nasleđivanje šablona Jinja2 može pomoći u tome. Na isti način na koji Flask-Bootstrap pruža osnovni predložak s osnovnim izgledom stranice, aplikacija može definirati svoj vlastiti osnovni predložak s uniformnim izgledom stranice koji uključuje navigacijsku traku i ostavlja sadržaj stranice da bude definiran u izvedenim predlošcima. . [Primjer 3-7](#) prikazuje šablonе/base.html, novi predložak koji nasleđuje od bootstrap/base.html i definira navigacijsku traku, ali je sam po sebi osnovni predložak drugog nivoa za druge šablone kao što su templates/user.html, predložci /404.html i templates/500.html.

Primjer 3-7. templates/base.html: osnovni predložak aplikacije sa navigacijskom trakom

```
{% proširuje "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation"> <div
class="container"> <div class="navbar-header">

    <button type="button" class="navbar-toggle" data-
    toggle="collapse" data-target=".navbar-collapse"> <span class="sr-
    only">Uključi navigaciju <span class="icon-bar"> <span
    class="icon-bar"> <span class="icon-bar"> </button> <a
    class="navbar -brand" href="/">Flasky</a>
```

```
</div>
<div class="navbar-collapse collapse">
    <ul class="none navbar-none">
        <li><a href="/">Početna</a></li> </
    ul> </div> </div> {% endblock %}

{% block content %}
<div class="container">{%
    block page_content %}{% endblock %}</div>
{% endblock %}
```

Blok sadržaja ovog predloška je samo element kontejnera `<div>` koji obavlja novi prazan blok pod nazivom `page_content`, koji izvedeni predlošci mogu definirati.

Predlošci aplikacije će sada naslijediti ovaj šablon umjesto direktno iz Flask-Bootstrapa. [Primjer 3-8](#) pokazuje koliko je jednostavno konstruirati prilagođenu stranicu greške koda 404 koja nasljeđuje od `templates/base.html`. Stranica za grešku 500 je slična i možete je pronaći u GitHub spremištu za aplikaciju.

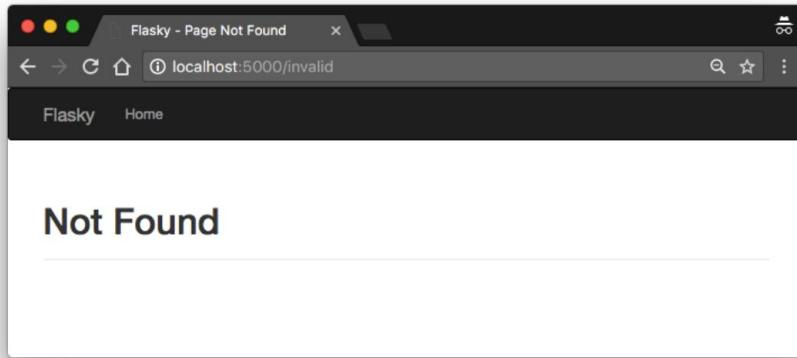
Primjer 3-8. `templates/404.html`: prilagođena stranica greške koda 404 koja koristi nasljeđivanje šablona

```
{% proširuje "base.html" %}

{% block title %}Flasky - Stranica nije pronađena{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Nije pronađeno</h1>
</div> {% endblock %}
```

Slika 3-2 prikazuje kako stranica o grešci izgleda u pretraživaču.



Slika 3-2. Stranica greške prilagođenog koda 404

Šablon templates/user.html sada se može pojednostaviti tako što će ga naslijediti od osnovnog predloška, kao što je prikazano u [primjeru 3-9](#).

Primjer 3-9. templates/user.html: pojednostavljeni predložak stranice koji koristi nasljeđivanje šablona

```
{% proširuje "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Zdravo, {{ name }}!</h1>
</div> {% endblock %}
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 3c da provjerite ovu verziju aplikacije.

Linkovi

Svaka aplikacija koja ima više od jedne rute će uvijek morati uključiti veze koje povezuju različite stranice, kao što je navigacijska traka.

Pisanje URL-ova kao linkova direktno u predlošku je trivijalno za jednostavne rute, ali za dinamičke rute s promjenjivim dijelovima može biti složenije napraviti URL-ove

pravo u šablonu. Također, URL-ovi napisani eksplisitno stvaraju neželjenu ovisnost o rutama definiranim u kodu. Ako se rute reorganiziraju, veze u predlošcima mogu pokvariti.

Da bi izbjegao ove probleme, Flask pruža pomoćnu funkciju `url_for()`, koja generiše URL-ove iz informacija pohranjenih u URL mapi aplikacije.

U svojoj najjednostavnijoj upotrebi, ova funkcija uzima naziv funkcije prikaza (ili naziv krajnje točke za rute definirane s `app.add_url_route()`) kao jedini argument i vraća njen URL. Na primjer, u trenutnoj verziji `hello.py` poziv `url_for('index')` bi vratio `/`, korijenski URL aplikacije. Pozivanje `url_for('index', _external=True)` bi umjesto toga vratio apsolutni URL, koji je u ovom primjeru `http://localhost:5000/`.



Relativni URL-ovi su dovoljni kada se generišu veze koje povezuju različite rute aplikacije. Apsolutni URL-ovi su neophodni samo za veze koje će se koristiti izvan web pretraživača, kao što je kod slanja linkova putem e-pošte.

Dinamički URL-ovi se mogu generirati pomoću `url_for()` proslijedivanjem dinamičkih dijelova kao argumenata ključne riječi. Naprimjer, `url_for('hello', _external=True)` bi vratio `http://localhost:5000/`

Argumenti ključne riječi poslati u `url_for()` nisu ograničeni na argumente koje koriste dinamičke rute. Funkcija će dodati sve argumente koji nisu dinamički u niz upita.

Na primjer, `url_for('user', name='john', page=2, version=1)` bi vratio `/user/john?page=2&version=1`.

Static Files

Web aplikacije nisu napravljene samo od Python koda i šablonu. Većina aplikacija takođe koristi statične datoteke kao što su slike, JavaScript izvorni fajlovi i CSS fajlovi koji su svi referencirani iz HTML koda u predlošcima.

Možda se sećate da kada je URL mapa aplikacije `hello.py` pregledana u [Poglavlju 2](#), u njoj se pojavio statički unos. Flask automatski podržava statičke datoteke dodavanjem posebne rute u aplikaciju definiranu kao `/static/<filename>`. Na primjer, poziv `url_for('static', filename='css/styles.css', _external=True)` bi vratio `http://localhost:5000/static/css/styles.css`.

U svojoj zadanoj konfiguraciji, Flask traži statičke datoteke u poddirektorijumu koji se zove `static` koji se nalazi u osnovnom folderu aplikacije. Datoteke se mogu organizirati u poddirektorijumima unutar ove mape ako želite. Kada server primi URL koji se preslikava na statičku rutu, on

generira odgovor koji uključuje sadržaj odgovarajuće datoteke u sistemu datoteka.

Primjer 3-10 pokazuje kako aplikacija može uključiti ikonu favicon.ico u osnovni predložak za preglednike da se prikazuju u adresnoj traci.

Primjer 3-10. templates/base.html: favicon definition

```
{% glava bloka %}
{{ super() }} <link
rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}" type="image/x-icon"> <link
rel="icon" href="{{ url_for('static', filename='favicon.ico') }}" type="image/x-icon"> {% endblock %}
```

Deklaracija ikone je umetnuta na kraju glavnog bloka. Obratite pažnju na to kako se super() koristi za očuvanje originalnog sadržaja bloka definiranog u osnovnim predlošcima.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 3d da provjerite ovu verziju aplikacije.

Lokalizacija datuma i vremena sa Flask-Moment

Rukovanje datumima i vremenom u web aplikaciji nije trivijalan problem kada korisnici rade u različitim dijelovima svijeta.

Serveru su potrebne uniformne vremenske jedinice koje su nezavisne od lokacije svakog korisnika, tako da se obično koristi koordinirano univerzalno vrijeme (UTC). Za korisnike, međutim, gledanje vremena izraženog u UTC-u može biti zbunjujuće, jer korisnici uvijek očekuju da vide datume i vremena prikazane u njihovom lokalnom vremenu i formatirane u skladu sa običajima njihovog regiona.

Elegantno rješenje koje omogućava serveru da radi isključivo u UTC-u je slanje ovih vremenskih jedinica u web pretraživač, gdje se one pretvaraju u lokalno vrijeme i renderiraju pomoću JavaScripta. Web pretraživači mogu mnogo bolje obaviti ovaj zadatak jer imaju pristup postavkama vremenske zone i lokalizacije na korisnikovom računaru.

Postoji odlična biblioteka otvorenog koda napisana u JavaScript-u koja prikazuje datume i vremena u pretraživaču pod nazivom [Moment.js](#). Flask-Moment je proširenje za Flask aplikacije koje čini integraciju Moment.js u Jinja2 šablone vrlo jednostavnom. Flask Moment se instalira sa pip:

```
(venv) $ pip install flask-moment
```

Ekstenzija se inicijalizira na sličan način kao i Flask-Bootstrap. Traženi kod je prikazan u [primjeru 3-11](#).

Primjer 3-11. hello.py: inicijaliziranje Flask-Momenta

```
from flask_moment import Moment
moment = Moment(app)
```

Flask-Moment zavisi od jQuery.js pored Moment.js. Ove dvije biblioteke moraju biti uključene negdje u HTML dokument - ili direktno, u kom slučaju možete odabrati koje verzije želite koristiti, ili putem pomoćnih funkcija koje pruža ekstenzija, koje upućuju na testirane verzije ovih biblioteka iz mreže za isporuku sadržaja (CDN). Budući da Bootstrap već uključuje jQuery.js, u ovom slučaju treba dodati samo Moment.js. [Primer 3-12](#) pokazuje kako se ova biblioteka učitava u blok skripti šablona, dok se takođe čuva originalni sadržaj bloka koji obezbeđuje osnovni šablon. Imajte na umu da budući da je ovo unaprijed definirani blok u osnovnom predlošku Flask-Bootstrap, lokacija u templates/base.html gdje je ovaj blok umetnut nije bitna.

Primjer 3-12. templates/base.html: uvoz biblioteke Moment.js

```
{% blok skripti %}
{{ super() }}
{{ moment.include_moment() }} {% endblock %}
```

Za rad sa vremenskim oznakama, Flask-Moment čini objekt momenta dostupnim predlošcima. [Primjer 3-13](#) demonstrira proslijđivanje varijable zvane current_time predlošku za renderiranje.

Primjer 3-13. hello.py: dodavanje varijable datuma i vremena

```
from datetime import datetime

@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

[Primjer 3-14](#) pokazuje kako se prikazuje ova varijabla šablona current_time .

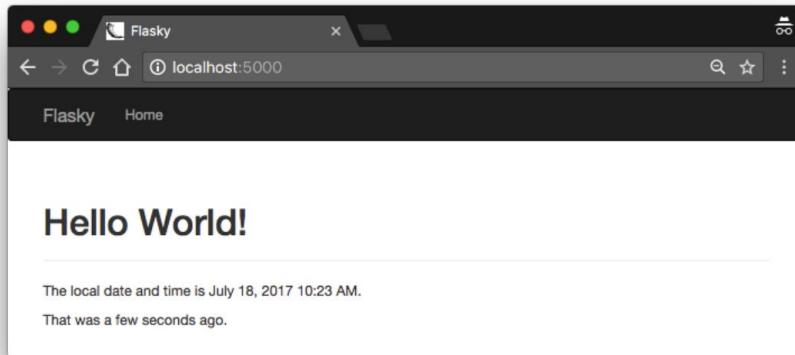
Primjer 3-14. templates/index.html: prikazivanje vremenske oznake sa Flask-Momentom

```
<p>Lokalni datum i vrijeme su {{ moment(current_time).format('LLL') }}.</p> <p>To je bilo
{{ moment(current_time).fromNow(refresh=True) }}</p>
```

Funkcija `format('LLL')` prikazuje datum i vrijeme prema postavkama vremenske zone i lokalizacije na klijentskom računalu. Argument određuje stil prikazivanja, od 'L' do 'LLLL' za četiri različita nivoa opširnosti. Funkcija `format()` također može prihvati dugu listu prilagođenih specifikacija formata.

Stil renderiranja `fromNow()` prikazan u drugom redu prikazuje relativnu vremensku oznaku i automatski je osvježava kako vrijeme prolazi. U početku će ova vremenska oznaka biti prikazana kao "prije nekoliko sekundi", ali opcija `refresh=True` će je održavati ažuriranom kako vrijeme prolazi, tako da ako ostavite stranicu otvorenu nekoliko minuta, vidjet ćete da se tekst mijenja u "prije minuti". .", zatim "prije 2 minute" i tako dalje.

Slika 3-3 pokazuje kako izgleda ruta `http://localhost:5000/` nakon što se dvije vremenske oznake dodaju u predložak `index.html`.



Slika 3-3. Stranica sa dvije vremenske oznake Flask-Moment



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti `git checkout 3e` da provjerite ovu verziju aplikacije.

Flask-Moment implementira metode `format()`, `fromNow()`, `fromTime()`, `calendar()`, `valueOf()` i `unix()` iz Moment.js. Konsultujte [Moment.js dokumentaciju](#) da saznote o svim opcijama formatiranja koje nudi ova biblioteka.



Flask-Moment prepostavlja da su vremenske oznake kojima ruke aplikacija na strani servera „naivni“ objekti datuma i vremena izraženi u UTC-u. Pogledajte dokumentaciju za [datetim paket](#) u standardnoj biblioteci za informacije o naivnim i svjesnim objektima datuma i vremena.

Vremenske oznake koje prikazuje Flask-Moment mogu se lokalizirati na mnoge jezike. Jezik se može odabrat u šablonu prenošenjem [dvoslovnog koda jezika](#) za funkciju `locale()`, odmah nakon što je Moment.js biblioteka uključena. Na primjer, evo kako konfigurirati Moment.js da koristi španski:

```
{% blok skripti %}  
{{ super() }}  
{{ moment.include_moment() }}  
{{ moment.locale('es') }} {% endblock  
%}
```

Uz sve tehnike o kojima se govori u ovom poglavlju, trebali biste biti u mogućnosti da napravite moderne web stranice prilagođene korisniku za svoju aplikaciju. Sljedeće poglavlje dotiče se aspekta šablona o kojima se još nije raspravljalo: kako komunicirati s korisnikom putem web obrazaca.

Web Forms

Predlošci s kojima ste radili u [Poglavlju 3](#) su jednosmjerni, u smislu da omogućavaju protok informacija od servera do korisnika. Za većinu aplikacija, međutim, postoji i potreba za informacijama koje teku u drugom smjeru, pri čemu korisnik daje podatke koje server prihvata i obrađuje.

Uz HTML, moguće je kreirati [web forme](#), u koje korisnici mogu unositi informacije. Podatke obrasca zatim web pretraživač šalje serveru, obično u obliku POST zahtjeva. Objekat zahtjeva Flask, uveden u [Poglavlju 2](#), izlaže sve informacije koje klijent šalje u zahtjevu i, posebno za POST zahtjeve koji sadrže podatke obrasca, omogućava pristup korisničkim informacijama putem `request.form`.

Iako je podrška pružena u Flaskovom objektu zahtjeva dovoljna za rukovanje web obrascima, postoji niz zadataka koji mogu postati zamorni i ponavljajući.

Dva dobra primjera su generiranje HTML koda za obrasce i provjera valjanosti dostavljenih podataka obrasca.

Flask -[WTF](#) proširenje čini rad s web obrascima mnogo ugodnjim iskustvom. Ovo proširenje je flask integracijski omotač oko okvira agnostičkih [WTForms](#) paket.

Flask-WTF i njegove zavisnosti mogu se instalirati pomoću pip-a:

```
(venv) $ pip install flask-wtf
```

Konguracija

Za razliku od većine drugih ekstenzija, Flask-WTF ne mora biti inicijaliziran na razini aplikacije, ali očekuje da aplikacija ima konfiguriran tajni ključ. Tajni ključ je niz sa bilo kojim slučajnim i jedinstvenim sadržajem koji se koristi kao ključ za šifriranje ili potpis za poboljšanje sigurnosti aplikacije na nekoliko načina. Flask koristi ovaj ključ da zaštitи sadržaj korisničke sesije od neovlaštenog pristupa. Trebali biste odabratи drugačiji tajni ključ u svakoj aplikaciji koju izgradite i pobrinuti se da ovaj niz niko ne zna. [Primjer 4-1](#) pokazuje kako konfigurirati tajni ključ u aplikaciji Flask.

[Primjer 4-1. hello.py: Flask-WTF konguracija](#)

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'teško pogoditi niz'
```

Rječnik app.config je mjesto opće namjene za pohranjivanje konfiguracijskih varijabli koje koristi Flask, ekstenzije ili sama aplikacija. Vrijednosti konfiguracije mogu se dodati objektu app.config koristeći standardnu sintaksu rječnika. Konfiguracijski objekt također ima metode za uvoz konfiguracijskih vrijednosti iz datoteke ili okruženja. Praktičniji način upravljanja konfiguracijskim vrijednostima za veću aplikaciju bit će razmotren u 7. [poglavlju](#).

Flask-WTF zahtijeva da se tajni ključ konfiguriše u aplikaciji jer je ovaj ključ dio mehanizma koji ekstenzija koristi za zaštitu svih obrazaca od napada krivotvorenja zahtjeva na više lokacija (CSRF). CSRF napad se događa kada zlonamjerna web stranica šalje zahtjeve aplikacijskom serveru na kojem je korisnik trenutno prijavljen. Flask-WTF generiše sigurnosne tokene za sve obrasce i pohranjuje ih u korisničkoj sesiji, koja je zaštićena kriptografskim potpisom generiranim iz tajni ključ.



Za dodatnu sigurnost, tajni ključ bi trebao biti pohranjen u varijablu okruženja umjesto da bude ugrađen u izvorni kod. Ova tehnika je opisana u [poglavlju](#) 7.

Form Classes

Kada koristite Flask-WTF, svaki web obrazac je predstavljen na serveru klasom koja nasljeđuje klasu FlaskForm. Klasa definiše listu polja u obrascu, od kojih je svako predstavljeno objektom. Svaki objekt polja može imati jedan ili više priloženih validatora. Validator je funkcija koja provjerava da li su podaci koje je korisnik dostavio ispravni.

Primjer 4-2 prikazuje jednostavan web obrazac koji ima tekstualno polje i dugme za slanje.

Primjer 4-2. hello.py: definicija klase forme

```
from flask_wtf import FlaskForm iz
wtforms import StringField, SubmitField iz
wtforms.validators import DataRequired

class NameForm(FlaskForm):
    name = StringField('Kako se zovete?', validators=[DataRequired()]) submit =
    SubmitField('Submit')
```

Polja u obrascu su definirana kao varijable klase, a svakoj varijabli klase je dodijeljen objekt povezan s tipom polja. U ovom primjeru, obrazac NameForm ima tekstualno polje koje se zove ime i dugme za slanje koje se zove submit. Klasa StringField predstavlja HTML <input> element sa atributom type="text". SubmitField klasa predstavlja HTML <input> element sa type="submit" atributom. Prvi argument konstruktorima polja je oznaka koja će se koristiti prilikom prikazivanja obrasca u HTML.

Opcioni argument validatora uključen u konstruktor StringField definiše listu provera koji će se primeniti na podatke koje je korisnik dostavio pre nego što budu prihvaćeni. DataRequired () validator osigurava da polje nije poslano prazno.



Osnovna klasa FlaskForm definirana je ekstenzijom Flask-WTF, tako da se uvozi iz flask_wtf. Polja i validatori se, međutim, uvoze direktno iz paketa WTForms.

Lista standardnih HTML polja koje podržava WTForms prikazana je u [Tabeli 4-1](#).

Tabela 4-1. WTForms standardna HTML polja

Vrsta polja	Opis
BooleanField	Potvrđni okvir sa vrijednostima Tačno i Netačno
TextField	Tekstualno polje koje prihvata vrijednost datetime.date u datom formatu
DateTimeField	Tekstualno polje koje prihvata vrijednost datetime.datetime u datom formatu
DecimalNoField	Tekstualno polje koje prihvata decimalnu.Decimalnu vrijednost Polje
FileField	za prijenos datoteke
HiddenField	Skriveno tekstualno polje
MultipleFileField	Polje za učitavanje više datoteka
FieldList	Lista polja datog tipa

Vrsta polja	Opis
FloatField	Tekstualno polje koje prihvata vrijednost s pomičnim zarezom
FormField	Obrazac je ugrađen kao polje u obrazac kontejnera
IntegerField	Tekstualno polje koje prihvata cijelobrojnu vrijednost
PasswordField	Polje za tekst lozinke
RadioField	Lista radio dugmadi
Odaberite Polje	Padajuća lista izbora
SelectMultipleField	Padajuća lista izbora sa višestrukim izborom
SubmitField	Dugme za podnošenje obrasca
StringField	Polje za tekst
TextAreaField	Višelinjsko tekstualno polje

Lista WTForms ugrađenih validatora prikazana je u [Tabeli 4-2.](#)

Tabela 4-2. WTForms validatori

Validator	Opis
DataRequired	Provjerava da li polje sadrži podatke nakon konverzije tipa
Email	Potvrđuje adresu e-pošte
Jednak	Uspoređuje vrijednosti dva polja; korisno kada se traži da se lozinka unese dva puta radi potvrde
InputRequired	Provjerava da li polje sadrži podatke prije konverzije tipa
IPAddress	Potvrđuje IPv4 mrežnu adresu
Dužina	Provjerava dužinu unesenog niza
MacAddress	Potvrđuje MAC adresu
NumberRange	Potvrđuje da je unesena vrijednost unutar numeričkog raspona
Opciono	Omogućava prazan unos u polju, preskačući dodatne validatore
Regexp	Provjerava unos regularnog izraza
URL	Potvrđuje URL
UUID	Potvrđuje UUID
AnyOf	Potvrđuje da je unos jedna od liste mogućih vrijednosti
Nijedna	Potvrđuje da unos nije nijedan od liste mogućih vrijednosti

HTML prikazivanje obrazaca

Polja obrasca su pozivi koji se, kada se pozovu iz šablona, prikazuju u HTML-u. Pod pretpostavkom da funkcija view prosjećuje instancu NameForm predlošku kao argument nazvan formu, predložak može generirati jednostavan HTML obrazac na sljedeći način:

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }} </form>
```

Imajte na umu da pored polja za ime i podnošenje, obrazac ima element form.hidden_tag(). Ovaj element definira dodatno polje obrasca koje je skriveno, a koristi ga Flask-WTF za implementaciju CSRF zaštite.

Naravno, rezultat iscrtavanja web obrasca na ovaj način je krajnje ogoljen. Svi argumenti ključne riječi dodani pozivima koji prikazuju polja pretvaraju se u HTML atribute za polje – tako, na primjer, možete dati id polja ili atribute klase i zatim definirati CSS stilove za njih:

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(id='my-text-field') }} {{ form.submit() }}
</form>
```

Ali čak i sa HTML atributima, napor potreban da bi se obrazac prikazao na ovaj način i da bi izgledao dobro je značajan, tako da je najbolje iskoristiti Bootstrap-ov vlastiti skup stilova obrasca kad god je to moguće. Ekstenzija Flask-Bootstrap pruža pomoćnu funkciju visokog nivoa koja prikazuje cijeli Flask-WTF obrazac koristeći Bootstrap unaprijed definirane stilove obrasca, sve jednim pozivom. Koristeći Flask-Bootstrap, prethodni obrazac se može prikazati na sljedeći način:

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

Direktiva uvoza radi na isti način kao i obične Python skripte i omogućava da se elementi šablona uvezu i koriste u mnogim predlošcima. Uvezena datoteka bootstrap/wtf.html definira pomoćne funkcije koje prikazuju Flask-WTF forme koristeći Bootstrap. Funkcija wtf.quick_form() uzima objekat obrasca Flask-WTF i prikazuje ga koristeći zadane Bootstrap stilove. Kompletan predložak za hello.py prikazan je u [primjeru 4-3](#).

Primjer 4-3. templates/index.html: korištenje Flask-WTF i Flask-Bootstrap za renderiranje obrasca

```
{% proširuje "base.html" %} {%
import "bootstrap/wtf.html" kao wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Zdravo, {{ name }}{{ else %}}Stranac{{ endif %}}!</h1> </div>
{{ wtf.quick_form(form) }} {% endblock %}
```

Područje sadržaja predloška sada ima dva odjeljka. Prvi dio je zaglavje stranice koje prikazuje pozdrav. Ovdje se koristi uslovni predložak. Uslovi u Jinja2 imaju format {% if uslov %}...{% else %}...{% endif %}. Ako se uvjet procijeni na Tačno, onda se ono što se pojavljuje između if i else direktiva dodaje u renderirani predložak. Ako se uvjet procijeni na False, onda se umjesto toga prikazuje ono što je između else i endif . Svrha ovoga je da prikaže Zdravo, {{ name }}! kada je definisana varijabla predloška imena ili string Zdravo, Stranger! kada nije. Drugi dio sadržaja prikazuje formu NameForm koristeći funkciju wtf.quick_form() .

Rukovanje obrascima u funkcijama prikaza

U novoj verziji hello.py, funkcija prikaza index() imat će dva zadatka. Prvo će prikazati obrazac, a zatim će primiti podatke obrasca koje je unio korisnik.

Primjer 4-4 prikazuje ažuriranu funkciju prikaza index() .

Primjer 4-4. hello.py: rukovati web formom pomoću metoda GET i POST zahtjeva

```
@app.route('/', methods=['GET', 'POST']) def index():

    ime = Nema
    form = NameForm()
    ako form.validate_on_submit(): ime
        = form.name.data
        " "
    form.name.data =
    return render_template('index.html', form=form, name=name)
```

Argument method dodan dekoratoru app.route govori Flasku da registruje funkciju pogleda kao rukovaoca za GET i POST zahtjeve u URL mapi. Kada metode nisu date, funkcija pregleda je registrirana za rukovanje samo GET zahtjevima.

Dodavanje POST -a na listu metoda je neophodno jer se podnošenjem obrasca mnogo lakše rukuje kao POST zahtjevima. Moguće je poslati obrazac kao GET zahtjev, ali kako GET zahtjevi nemaju tijelo, podaci se dodaju URL-u kao string upita i postaju vidljivi u adresnoj traci pretraživača. Iz ovog i nekoliko drugih razloga, podnošenje obrasca se gotovo univerzalno obavlja kao POST zahtjevi.

Varijabla lokalnog imena koristi se za čuvanje imena primljenog iz obrasca kada je dostupno; kada ime nije poznato, varijabla se inicijalizira na None. Funkcija prikaza kreira instancu klase NameForm koja je prethodno prikazana da predstavlja formu. Metoda validate_on_submit() obrasca vraća True kada je obrazac dostavljen i kada su svi validatori polja prihvatali podatke. U svim ostalim slučajevima, validate_on_submit() vraća False. Povratna vrijednost ove metode efektivno služi za određivanje da li je obrazac potrebno prikazati ili obraditi.

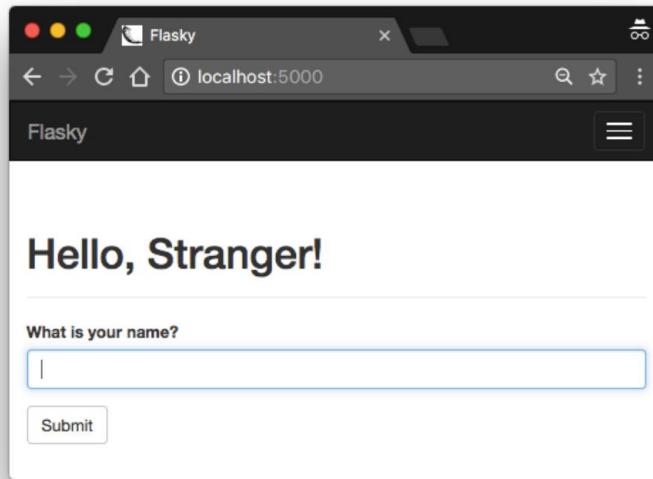
Kada korisnik prvi put dođe do aplikacije, server će primiti GET zahtjev bez podataka obrasca, tako da će validate_on_submit() vratiti False. Tijelo if naredbe će biti preskočeno i zahtjevom će se rukovati prikazivanjem šablona, koji kao argument dobija objekt forme i promjenjivu name postavljenu na None . Korisnici će sada vidjeti obrazac prikazan u pretraživaču.

Kada korisnik pošalje obrazac, server prima POST zahtjev sa podacima. Poziv validate_on_submit() poziva validator DataRequired() koji je pridodat polju imena. Ako ime nije prazno, validator ga prihvata i validate_on_submit() vraća True. Sada je ime koje je unio korisnik dostupan kao atribut podataka polja. Unutar tijela if naredbe, ovo ime se dodjeljuje varijabli lokalnog imena i polje obrasca se briše postavljanjem tog atributa podataka na prazan niz, tako da je polje prazno kada se obrazac ponovo prikaže na stranici. Poziv render_template() u posljednjem redu prikazuje predložak, ali ovaj put argument name sadrži ime iz obrasca, tako da će pozdrav biti personaliziran.

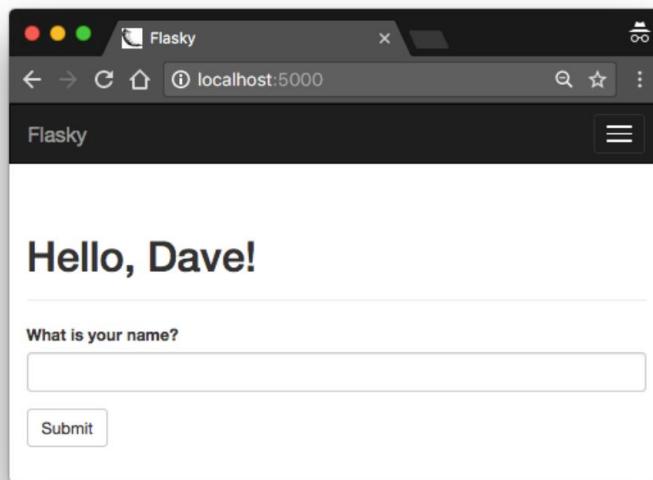


Ako ste klonirali Git spremište aplikaciju na GitHub-u, možete pokrenuti git checkout 4a da provjerite ovu verziju aplikacije.

Slika 4-1 pokazuje kako obrazac izgleda u prozoru pretraživača kada korisnik inicijalno uđe na stranicu. Kada korisnik unese ime, aplikacija odgovara personalizovanim pozdravom. Obrazac se i dalje pojavljuje ispod njega, tako da ga korisnik može poslati više puta s različitim imenima ako želi. **Slika 4-2** prikazuje aplikaciju u ovom stanju.

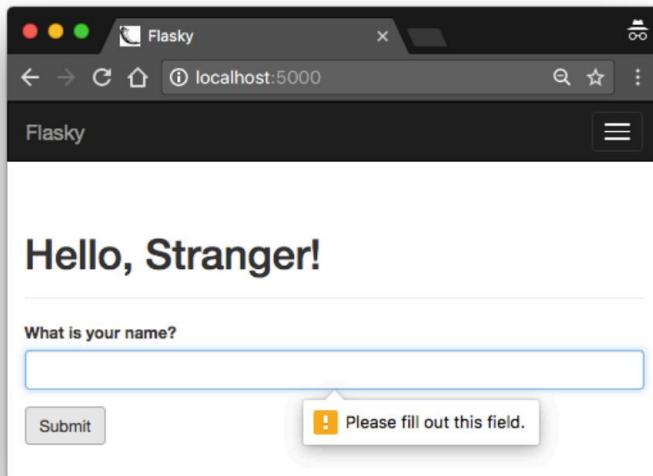


Slika 4-1. Flask-WTF web obrazac



Slika 4-2. Web obrazac za podnošenje

Ako korisnik pošalje obrazac sa praznim imenom, DataRequired() validator hvata grešku, kao što se vidi na [slici 4-3](#). Obratite pažnju na to koliko funkcija se automatski pruža. Ovo je sjajan primjer snage koju dobro dizajnirane ekstenzije poput Flask-WTF i Flask-Bootstrap mogu dati vašoj aplikaciji.



Slika 4-3. Web obrazac aer nije uspio validator

Preusmjeravanja i korisničke sesije

Poslednja verzija hello.py ima problem upotrebljivosti. Ako unesete svoje ime i pošaljete ga, a zatim kliknete na dugme za osvežavanje u pretraživaču, verovatno ćete dobiti nejasno upozorenje koje traži potvrdu pre ponovnog podnošenja obrasca. To se dešava zato što pretraživač ponavljaju posljednji zahtjev koji su poslali kada se od njih traži da osvježe stranicu. Kada je posljednji poslani zahtjev POST zahtjev s podacima obrasca, osvježavanje bi izazvalo duplo podnošenje obrasca, što u gotovo svim slučajevima nije željena radnja. Iz tog razloga, pretraživač traži potvrdu od korisnika.

Mnogi korisnici ne razumiju ovo upozorenje pretraživača. Shodno tome, smatra se dobrom praksom da web aplikacije nikada ne ostavljaju POST zahtjev kao posljednji zahtjev koji je poslao pretraživač.

Ovo se postiže tako što se na POST zahtjeve odgovara preusmjeravanjem umjesto normalnog odgovora. Preusmjeravanje je poseban tip odgovora koji sadrži URL umjesto stringa sa HTML kodom. Kada pretraživač primi odgovor za preusmjeravanje, izdaje a

GET zahtjev za URL za preusmjeravanje, a to je stranica koju prikazuje. Stranica može potrajati još nekoliko milisekundi da se učita zbog drugog zahtjeva koji se mora poslati serveru, ali osim toga, korisnik neće vidjeti nikakvu razliku. Sada je posljednji zahtjev GET, tako da komanda za osvježavanje radi kako se očekuje. Ovaj trik je poznat kao obrazac Post/Redirect/Get.

Ali ovaj pristup donosi drugi problem. Kada aplikacija obrađuje POST zahtjev, ima pristup imenu koje je korisnik unio u form.name.data, ali čim se taj zahtjev završi podaci obrasca se gube. Budući da se POST zahtjevom rukuje s preusmjeravanjem, aplikacija treba pohraniti ime tako da ga preusmjereni zahtjev može imati i koristiti ga za izgradnju stvarnog odgovora.

Aplikacije mogu "pamtiti" stvari iz jednog zahtjeva u drugi tako što ih pohranjuju u korisničku sesiju, privatno skladište koje je dostupno svakom povezanom klijentu. Korisnička sesija je predstavljena u poglavljiju 2 kao jedna od varijabli povezanih s kontekstom zahtjeva. Zove se sesija i pristupa se kao standardnom Python rječniku.



Podrazumevano, korisničke sesije se pohranjuju u kolačiće na strani klijenta koji su kriptografski potpisani pomoću konfigurisanog tajnog ključa. Svako mijenjanje sadržaja kolačića učinilo bi potpis nevažećim, čime bi poništila sesiju.

Primjer 4-5 prikazuje novu verziju funkcije prikaza index() koja implementira preusmjeravanja i korisničke sesije.

Primjer 4-5. hello.py: preusmjeravanja i korisničke sesije

```
iz flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST']) def
index(): form = NameForm()
    if form.validate_on_submit():
        session['name'] =
        form.name.data return redirect(url_for('index'))
        return render_template('index.html',
            form=form, name=session.get('name'))
```

U prethodnoj verziji aplikacije, varijabla lokalnog imena je korištena za pohranjivanje imena koje je korisnik unio u obrazac. Ta varijabla se sada postavlja u korisničku sesiju kao session['name'] tako da se pamti nakon zahtjeva.

Zahtjevi koji dolaze s važećim podacima obrasca sada će završiti pozivom redirect(), pomoćne funkcije Flask koja generiše HTTP odgovor preusmjeravanja. Funkcija redirect() uzima URL na koji se preusmjerava kao argument. URL za preusmjeravanje koji se koristi u ovom slučaju je korijenski URL, tako da je odgovor mogao biti napisan konciznije kao redirect('/'), ali umjesto toga se koristi Flaskova funkcija za generiranje URL-a url_for(), predstavljena u poglavlju 3.

Prvi i jedini potrebni argument za url_for() je ime krajnje tačke, interno ime koje svaka ruta ima. Podrazumevano, krajnja tačka rute je naziv funkcije pregleda koja joj je pridružena. U ovom primjeru, funkcija pogleda koja rukuje korijenskim URL-om je index(), tako da je ime dato url_for() index.

Posljednja promjena je u funkciji render_template() , koja sada dobiva argument name direktno iz sesije koristeći session.get('name'). Kao i kod redovnih rečnika, korišćenje get() za traženje ključa rečnika izbegava izuzetak za ključeve koji nisu pronađeni. Metoda get() vraća zadalu vrijednost None za ključ koji nedostaje.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 4b da provjerite ovu verziju aplikacije.

Sa ovom verzijom aplikacije, možete vidjeti da osvježavanje stranice u vašem pretraživaču uvijek rezultira očekivanim ponašanjem.

Poruka treperi

Ponekad je korisno dati korisniku ažuriranje statusa nakon što je zahtjev završen. Ovo može biti poruka potvrde, upozorenje ili greška. Tipičan primjer je kada pošaljete obrazac za prijavu na web stranicu s greškom, a server odgovori tako što ponovo prikaže obrazac za prijavu s porukom iznad njega koja vas obaveštava da je vaše korisničko ime ili lozinka nevažeća.

Flask uključuje ovu funkcionalnost kao osnovnu funkciju. [Primjer 4-6](#) pokazuje kako se funkcija flash() može koristiti u tu svrhu.

Primjer 4-6. hello.py: flash poruke

```
iz flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', methods=['GET', 'POST']) def
index():
    form = NameForm()
    if
        form.validate_on_submit():

            old_name = session.get('name') ako
            old_name nije None i old_name != form.name.data: flash('Izgleda da
                ste promjenili ime!') session['name'] = form.name.data: return
            redirect(url_for('index')) return render_template('index.html',
                form = form, name = session.get('name'))
```

U ovom primjeru, svaki put kada se podnese ime, ono se upoređuje s imenom pohranjenim u korisničkoj sesiji, koje će tamo biti stavljeno tokom prethodnog podnošenja istog obrasca. Ako su dva imena različita, funkcija flash() se poziva s porukom koja se prikazuje na sljedećem odgovoru koji se šalje nazad klijentu.

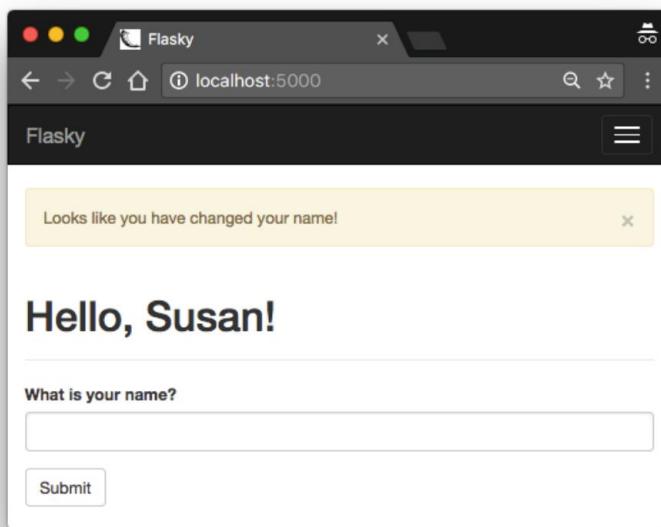
Pozivanje flash() nije dovoljno da bi se poruke prikazale; predlošci koje koristi aplikacija trebaju za prikazivanje ovih poruka. Najbolje mjesto za renderiranje flash poruka je osnovni šablon, jer će to omogućiti ove poruke na svim stranicama. Flask čini funkciju get_flashed_messages() dostupnom predlošcima za preuzimanje poruka i njihovo prikazivanje, kao što je prikazano u [primjeru 4-7](#).

Primjer 4-7. templates/base.html: prikazivanje flashed poruka

```
{% block content %}
<div class="container">
    {% for poruka in get_flashed_messages() %} <div
        class="alert alert-warning">
            <button type="button" class="close" data-dismiss="alert">&times;</button> {{ poruka }} </
        div> {% endfor %}

    {% block page_content %}{% endblock %} </div>
{% endblock %}
```

U ovom primjeru, poruke se prikazuju korištenjem Bootstrap-ovih CSS stilova upozorenja za poruke upozorenja (jedna je prikazana na [slici 4-4](#)).



Slika 4-4. Treperi poruka

Petlja se koristi jer bi moglo biti više poruka u redu za prikaz, jedna za svaki put kada je flash() pozvan u prethodnom ciklusu zahtjeva. Poruke koje su preuzete iz get_flashed_messages() neće biti vraćene sledeći put kada se ova funkcija pozove, tako da se fleš poruke pojavljuju samo jednom i zatim se odbacuju.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 4c da provjerite ovu verziju aplikacije.

Mogućnost prihvaćanja podataka od korisnika putem web obrazaca je značajka koju zahtijeva većina aplikacija, a isto tako i mogućnost pohranjivanja tih podataka u trajno skladište. Korištenje baza podataka sa Flaskom je tema sljedećeg poglavlja.

POGLAVLJE 5

Baze podataka

Baza podataka pohranjuje podatke aplikacije na organiziran način. Aplikacija zatim postavlja upite kako bi dohvatila određene dijelove podataka prema potrebi. Najčešće korišćene baze podataka za web aplikacije su one zasnovane na relacionom modelu, koje se nazivaju i SQL baze podataka u odnosu na jezik strukturiranih upita koji koriste.

Ali poslednjih godina, baze podataka orijentisane na dokumente i baze podataka ključ-vrednost, neformalno poznate zajedno kao NoSQL baze podataka, postale su popularne alternative.

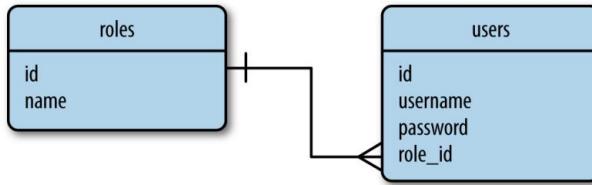
SQL baze podataka

Relacijske baze podataka pohranjuju podatke u tablice, koje modeliraju različite entitete u domenu aplikacije. Na primjer, baza podataka za aplikaciju za upravljanje narudžbama će vjerovatno imati tablice kupaca, proizvoda i narudžbi .

Tabela ima fiksni broj kolona i promjenjiv broj redova. Stupci definiraju atribute podataka entiteta predstavljenog tablicom. Na primjer, tabela kupaca će imati kolone kao što su ime, adresa, telefon i tako dalje. Svaki red u tabeli definira stvarni element podataka koji dodjeljuje vrijednosti nekim ili svim stupcima.

Tabele imaju poseban stupac koji se zove primarni ključ, koji sadrži jedinstveni identifikator za svaki red pohranjen u tablici. Tabele također mogu imati kolone koje se nazivaju strani ključevi, koji upućuju na primarni ključ reda u istoj ili drugoj tabeli. Ove veze između redova nazivaju se odnosima i temelj su modela relacijske baze podataka.

Slika 5-1 prikazuje dijagram jednostavne baze podataka s dvije tablice koje pohranjuju korisnike i korisničke uloge. Linija koja povezuje dvije tabele predstavlja odnos između tabela.



Slika 5-1. Primjer relacijske baze podataka

Ovaj grafički stil predstavljanja strukture baze podataka naziva se dijagram odnosa entiteta. U ovom prikazu, okviri predstavljaju tabele baze podataka, prikazujući liste atributa ili kolona tabele. Tabela uloga pohranjuje listu svih mogućih korisničkih uloga, od kojih je svaka identificirana jedinstvenom vrijednošću ID -a – primarnim ključem tablice. Tabela korisnika sadrži listu korisnika, svaki sa svojim jedinstvenim ID -om . Pored id primarnih ključeva, tabela uloga ima kolonu imena , a tabela korisnika ima stupce korisničkog imena i lozinke .

Stupac role_id u tabeli korisnika je strani ključ. Linija koja povezuje stupce roles.id i users.role_id predstavlja odnos između dvije tablice. Simboli pričvršćeni za liniju na svakom kraju ukazuju na kardinalnost odnosa. Na strani roles.id , linija je prikazana kao "jedan", dok je na strani users.role_id predstavljeno "mnogo". Ovo prikazuje odnos jedan-prema-više, što ukazuje da svaki red iz tabele uloga može biti povezan sa mnogo redova iz

tabela korisnika .

Kao što se vidi u primeru, relacione baze podataka efikasno skladište podatke i izbegavaju dupliranje. Preimenovanje korisničke uloge u ovoj bazi podataka je jednostavno jer imena uloga postoje na jednom mjestu. Odmah nakon što se naziv uloge promijeni u tabeli uloga , svi korisnici koji imaju role_id koji upućuje na promijenjenu ulogu vidjet će ažuriranje.

S druge strane, dijeljenje podataka u više tabele može biti komplikacija. Izrada liste korisnika s njihovim ulogama predstavlja mali problem, jer korisnici i korisničke uloge moraju biti pročitani iz dvije tabele i spojeni prije nego što se mogu predstaviti zajedno. Motori relacionih baza podataka pružaju podršku za izvođenje operacija spajanja između tabela kada je to potrebno.

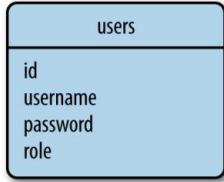
NoSQL baze podataka

Baze podataka koje ne prate relacioni model opisan u prethodnom odeljku zajednički se nazivaju NoSQL bazama podataka. Jedna uobičajena organizacija za NoSQL baze podataka koristi kolekcije umjesto tabele i dokumente umjesto zapisa.

NoSQL baze podataka dizajnirane su na način da otežava spajanje, tako da većina njih uopće ne podržava ovu operaciju. Za NoSQL bazu podataka strukturiranu kao na [slici 5-1](#),

listanje korisnika sa njihovim ulogama zahteva da sama aplikacija izvrši operaciju pridruživanja čitanjem polja role_id svakog korisnika, a zatim pretražujući tabelu uloga za to.

Prikladniji dizajn za NoSQL bazu podataka prikazan je na [slici 5-2](#). Ovo je rezultat primjene operacije zvane denormalizacija, koja smanjuje broj tablica na račun duplicitiranja podataka.



Slika 5-2. Primjer NoSQL baze podataka

Baza podataka sa ovom strukturu ima ime uloge eksplicitno pohranjeno sa svakim korisnikom. Preimenovanje uloge tada se može pokazati kao skupa operacija koja može zahtijevati ažuriranje velikog broja dokumenata.

Ali nisu sve loše vijesti s NoSQL bazama podataka. Dupliciranje podataka omogućava brže postavljanje upita. Navođenje korisnika i njihovih uloga je jednostavno jer nisu potrebna nikakva pridruživanja.

SQL ili NoSQL?

SQL baze podataka su izvrsne u skladištenju strukturiranih podataka u efikasnom i kompaktnom obliku. Ove baze podataka se trude da očuvaju konzistentnost, čak i u slučaju nestanka struje ili hardverskih kvarova. Paradigma koja omogućava relacionim bazama podataka da dostignu ovaj visok nivo pouzdanosti naziva se **ACID**, što je skraćenica za atomičnost, konzistentnost, izolaciju i trajnost. NoSQL baze podataka opuštaju neke od ACID zahtjeva i kao rezultat toga ponekad mogu dobiti prednost u performansama.

Potpuna analiza i poređenje tipova baza podataka je izvan okvira ove knjige. Za male i srednje aplikacije, i SQL i NoSQL baze podataka su savršeno sposobne i imaju praktično jednake performanse.

Python Database Frameworks

Python ima pakete za većinu motora baza podataka, kako otvorenog koda, tako i komercijalnih. Flask ne postavlja ograničenja na to koji paketi baze podataka se mogu koristiti, tako da možete raditi sa MySQL, Postgres, SQLite, Redis, MongoDB, CouchDB ili DynamoDB ako vam je bilo koji od ovih omiljenih.

Kao da to nije dovoljan izbor, postoji i niz paketa slojeva apstrakcije baze podataka, kao što su SQLAlchemy ili MongoEngine, koji vam omogućavaju da radite na višem nivou s regularnim Python objektima umjesto entiteta baze podataka kao što su tabele, dokumenti ili jezici upita.

Postoji niz faktora koje treba procijeniti prilikom odabira okvira baze podataka:

Lakoća

upotrebe Prilikom upoređivanja direktnih mehanizama baze podataka sa slojevima apstrakcije baze podataka, druga grupa jasno pobjeđuje. Apstraktioni slojevi, koji se takođe nazivaju objektno-relacioni maperi (ORM) ili objektno-dokumentni maperi (ODM), obezbeđuju transparentnu konverziju objekata orientisanih operacija visokog nivoa u instrukcije baze podataka niskog nivoa.

Performanse

Pretvorbe koje ORM-ovi i ODM-ovi moraju učiniti da prevedu iz domene objekta u domenu baze podataka imaju prekomjerne troškove. U većini slučajeva, kazna za učinak je zanemarljiva, ali ne mora uvijek biti. Generalno, povećanje produktivnosti dobijeno sa ORM-ovima i ODM-ovima daleko nadmašuje minimalnu degradaciju performansi, tako da ovo nije valjan argument da se ORM-ovi i ODM-ovi potpuno odbace.

Ono što ima smisla je odabrati sloj apstrakcije baze podataka koji pruža opcioni pristup osnovnoj bazi podataka u slučaju da je potrebno optimizirati specifične operacije implementirajući ih direktno kao instrukcije izvorne baze podataka.

Prenosivost

Izbori baze podataka dostupni na vašim razvojnim i proizvodnim platformama moraju se uzeti u obzir. Na primjer, ako planirate hostirati svoju aplikaciju na platformi u oblaku, trebali biste saznati koje izbore baze podataka nudi ova usluga.

Drugi aspekt prenosivosti odnosi se na ORM-ove i ODM-ove. Iako neki od ovih okvira obezbeđuju sloj apstrakcije za jednu mašinu baze podataka, drugi apstrahuju još više i pružaju izbor mehanizama baze podataka—svima njima se može pristupiti sa istim objektno orientisanim interfejsom. Najbolji primjer za to je SQLAlchemy ORM, koji podržava listu motora za relacijske baze podataka uključujući popularni MySQL, Postgres i SQLite.

Flask integracija

Odabir okvira koji ima integraciju sa Flaskom nije apsolutno obavezan, ali će vas uštedjeti od potrebe da sami pišete integracijski kod. Integracija flask-a mogla bi pojednostaviti konfiguraciju i rad, tako da bi trebalo dati prednost korištenju paketa koji je posebno dizajniran kao proširenje Flask-a.

Na osnovu ovih ciljeva, odabrani okvir baze podataka za primjere u ovoj knjizi bit će [Flask-SQLAlchemy](#), omotač proširenja Flask za [SQLAlchemy](#).

Upravljanje bazom podataka sa Flask-SQLAlchemy

Flask-SQLAlchemy je Flask ekstenzija koja pojednostavljuje upotrebu SQLAlchemy unutar Flask aplikacija. SQLAlchemy je moćan okvir relacijske baze podataka koji podržava nekoliko backendova baze podataka. Nudi ORM na visokom nivou i pristup niskom nivou izvornoj SQL funkcionalnosti baze podataka.

Kao i većina drugih ekstenzija, Flask-SQLAlchemy se instalira sa pip:

```
(venv) $ pip install flask-sqlalchemy
```

U Flask-SQLAlchemy, baza podataka je navedena kao URL. **Tabela 5-1** navodi format URL-ova za tri najpopularnija mehanizma baza podataka.

Tabela 5-1. URL-ovi baze podataka Flask-SQLAlchemy

Database engine	URL
MySQL	mysql://username:password@hostname/database
Postgres	postgresql://username:password@hostname/database
SQLite (Linux, macOS)	sqlite:///absolute/path/to/database
SQLite (Windows)	sqlite:///c:/absolute/path/to/database

U ovim URL-ovima, ime hosta se odnosi na server koji hostuje uslugu baze podataka, što može biti lokalni ili udaljeni server. Serveri baza podataka mogu ugostiti nekoliko baza podataka, tako da baza podataka označava ime baze podataka koju treba koristiti. Za baze podataka kojima je potrebna autentifikacija, korisničko ime i lozinka su korisnički akreditivi baze podataka.



SQLite baze podataka nemaju server, tako da su ime hosta, korisničko ime i lozinka izostavljeni, a baza podataka je ime datoteke na disku za bazu podataka.

URL baze podataka aplikacije mora biti konfiguriran kao ključ `SQLALCHEMY_DATABASE_URI` u Flask konfiguracijskom objektu. Dokumentacija Flask-SQLAlchemy također predlaže postavljanje ključa `SQLALCHEMY_TRACK_MODIFICATIONS` na `False` za korištenje manje memorije osim ako nisu potrebni signali za promjene objekata. Konsultirajte Flask SQLAlchemy dokumentaciju za informacije o drugim opcijama konfiguracije.

Primjer 5-1 pokazuje kako inicijalizirati i konfigurirati jednostavnu SQLite bazu podataka.

Primjer 5-1. hello.py: konfiguracija baze podataka

```
import os
from flask_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))


```

```

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
    'sqlite:/// + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Netačno

db = SQLAlchemy(app)

```

db objekt instanciran iz klase SQLAlchemy predstavlja bazu podataka i pruža pristup svim funkcionalnostima Flask-SQLAlchemy.

Model Denition

Termin model se koristi kada se odnosi na trajne entitete koje koristi aplikacija. U kontekstu ORM-a, model je tipično Python klasa s atributima koji odgovaraju stupcima odgovarajuće tablice baze podataka.

Instanca baze podataka iz Flask-SQLAlchemy pruža osnovnu klasu za modele kao i skup pomoćnih klasa i funkcija koje se koriste za definiranje njihove strukture. Tabele uloga i korisnika sa [slike 5-1](#) mogu se definirati kao modeli Uloga i Korisnik kao što je prikazano u [Primjeru 5-2](#).

Primjer 5-2. hello.py: Definicija modela uloge i korisnika

```

class Uloga(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Uloga %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    korisnicko_ime = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<Korisnik %r>' % self.username

```

Varijabla klase `__tablename__` definira ime tablice u bazi podataka. Flask SQLAlchemy dodjeljuje zadano ime tablice ako je `__tablename__` izostavljeno, ali ta zadana imena ne sljede popularnu konvenciju korištenja množine za nazive tablica, pa je najbolje da se tablice imenuju eksplisitno. Preostale varijable klase su atributi modela, definirani kao instance klase `db.Column`.

Prvi argument dat konstruktoru db.Column je tip stupca baze podataka i atribut modela. **Tabela 5-2** navodi neke od tipova kolona koji su dostupni, zajedno sa Python tipovima koji se koriste u modelu.

Tabela 5-2. Najčešći tipovi kolona SQLAlchemy

Unesite ime	Python tip	Opis
Integer	int	Regularni cijeli broj, obično 32 bita
SmallInteger	int	Cijeli broj kratkog dometa, obično 16 bita
BigInteger	int ili long	Neograničen cijeli broj preciznosti
Float	float	Broj s pomičnim zarezom
Numerički	decimalni.Decimalni	Broj fiksne tačke
String	str	String promjenjive dužine
Tekst	str	Niz promjenjive dužine, optimiziran za veliku ili neograničenu dužinu
Unicode	unikod	Unicode string promjenjive dužine
UnicodeText	unikod	Unicode string promjenjive dužine, optimizovan za veliku ili neograničenu dužinu
Boolean	bool	Boolean vrijednost
Datum	datetime.date	Vrijednost datuma
Vrijeme	datetime.time	Vremenska vrijednost
Datum i vrijeme	datetime.datetime	Vrijednost datuma i vremena
Interval	datetime.timedelta	Vremenski interval
Enum	str	Lista vrijednosti niza
PickleType	Bilo koji Python objekt	Automatska serijalizacija Pickle
LargeBinary	str	Binarna mrlja

Preostali argumenti db.Column specificiraju opcije konfiguracije za svaki atribut. **Tabela 5-3** navodi neke od dostupnih opcija.

Tabela 5-3. Najčešće opcije SQLAlchemy stupca

Naziv opcije	Opis
primarni_ključ	Ako je postavljeno na Tačno, kolona je primarni ključ tabele.
jedinstven	Ako je postavljeno na Tačno, nemojte dozvoliti duple vrijednosti za ovu kolonu.
index	Ako je postavljeno na Tačno, kreirajte indeks za ovu kolonu kako bi upiti bili efikasniji.
nullable	Ako je postavljeno na Tačno, dozvolite prazne vrijednosti za ovu kolonu. Ako je postavljeno na False, kolona neće dozvoliti null vrijednosti.
default	Definirajte zadatu vrijednost za kolonu.



Flask-SQLAlchemy zahtijeva da svi modeli definiraju kolonu primarnog ključa, koja se obično naziva id.

Iako to nije striktno neophodno, ova dva modela uključuju metodu `_repr_()` koja im daje čitljivu reprezentaciju stringova koja se može koristiti u svrhu otklanjanja grešaka i testiranja.

Odnosi

Relacijske baze podataka uspostavljaju veze između redova u različitim tabelama korištenjem relacija. Relacijski dijagram na [slici 5-1](#) izražava jednostavan odnos između korisnika i njihovih uloga. Ovo je odnos jedan prema više između uloga i korisnika, jer jedna uloga može pripadati mnogim korisnicima, ali svaki korisnik može imati samo jednu ulogu.

Primjer 5-3 pokazuje kako je odnos jedan-prema-više na [slici 5-1](#) predstavljen u klasama modela.

Primjer 5-3. hello.py: odnosi u modelima baze podataka

```
class Uloga(db.Model): #
    korisnici =
        db.relationship('User', backref='role')

class User(db.Model): #
    role_id =
        db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Kao što se vidi na [slici 5-1](#), odnos povezuje dva reda upotrebom stranog ključa. Stupac `role_id` koji je dodan modelu korisnika definiran je kao strani ključ i koji uspostavlja odnos. Argument `'roles.id'` za `db.ForeignKey()` specificira da stupac treba tumačiti kao da ima vrijednosti id -a iz redova u tabeli uloga .

Atribut korisnika dodan modelu Uloga predstavlja objektno orijentisani pogled na odnos, gledano sa strane „jedan“. S obzirom na instancu klase Role, atribut users će vratiti listu korisnika povezanih s tom ulogom (tj. strana „mnogo“). Prvi argument za `db.relationship()` pokazuje koji je model na drugoj strani odnosa. Klasa modela se može dati kao string ako je klasa definirana kasnije u modulu.

Argument `backref` za `db.relationship()` definira obrnuti smjer odnosa, dodavanjem atributa uloge korisničkom modelu. Ovaj atribut se može koristiti

na bilo kojoj instanci korisnika umjesto stranog ključa role_id za pristup modelu uloge kao objektu.

U većini slučajeva db.relationship() može sam locirati strani ključ odnosa, ali ponekad ne može odrediti koji stupac koristiti kao strani ključ. Na primjer, ako model korisnika ima dva ili više stupaca definiranih kao strani ključevi uloge , tada SQLAlchemy ne bi znao koji od ta dva treba koristiti. Kad god je konfiguracija stranog ključa dvosmislena, potrebno je dati dodatne argumente db.relationship() . **Tabela 5-4** navodi neke od uobičajenih opcija konfiguracije koje se mogu koristiti za definiranje odnosa.

Tabela 5-4. Uobičajene opcije SQLAlchemy odnosa

Naziv opcije	Opis Dodajte
backref	povratnu referencu u drugi model u odnosu.
primaryjoin	Eksplicitno navedite uvjet spajanja između dva modela. Ovo je neophodno samo za dvosmislene odnose.
lujen	Odredite kako će se povezane stavke učitati. Moguće vrijednosti su odabранe (stavke se učitavaju na zahtjev prvi put kada im se pristupi), neposredne (stavke se učitavaju kada se izvorni objekt učita), spojene (stavke se učitavaju odmah, ali kao spoj), potupit (stavke se učitavaju odmah , ali kao potupit), noload (stavke se nikada ne učitavaju) i dinamički (umjesto učitavanja stavki, daje se upit koji ih može učitati).
lista korisnika	Ako je postavljeno na False, koristite skalar umjesto liste.
Poredak po	Navedite redoslijed koji se koristi za stavke u odnosu.
sekundarno	Odredite ime tablice asocijacije koju ćete koristiti u relacijama mnogo-prema-više.
secondaryjoin	Specificirajte sekundarni uvjet spajanja za mnogo-prema-mnogo relacije kada ga SQLAlchemy ne može sam odrediti.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 5a da provjerite ovu verziju aplikacije.

Postoje i drugi tipovi odnosa osim jedan-prema-više. Odnos jedan-na-jedan može se izraziti na isti način kao jedan-prema-više, kao što je ranije opisano, ali sa opcijom liste korisnika postavljenom na False unutar db.relationship () definicije tako da strana "mnogo" postaje "jedna strana. Odnos više-prema-jedan također se može izraziti kao jedan-prema-više ako su tablice obrnute, ili se može izraziti sa stranim ključem i definicijom db.relationship() na strani "mnogo". Najsloženiji tip odnosa, mnogo-prema-više, zahtijeva dodatnu tablicu koja se zove asocijacijska ili spojna tablica. **U 12. poglavljju** ćete naučiti o odnosima mnogo-prema-više .

Operacije baze podataka

Modeli su sada u potpunosti konfigurirani prema dijagramu baze podataka na [slici 5-1](#) i spremni su za korištenje. Najbolji način da naučite kako raditi s ovim modelima je u Python ljustici. Sljedeći odeljci će vas provesti kroz najčešće operacije baze podataka u ljustici koja je pokrenuta komandom flask shell . Prije nego što koristite ovu naredbu, uvjerite se da je varijabla okruženja FLASK_APP postavljena na hello.py, kao što je prikazano u [poglavlju 2](#).

Kreiranje tabela Prva

stvar koju treba uraditi je da uputite Flask-SQLAlchemy da kreira bazu podataka zasnovanu na klasama modela. Funkcija db.create_all() locira sve podklase db.Model i kreira odgovarajuće tabele u bazi podataka za njih:

```
(venv) $ flask shell >>>
iz hello import db >>>
db.create_all()
```

Ako provjerite direktorij aplikacije, sada ćete tamo vidjeti novu datoteku pod nazivom data.sqlite, ime koje je dato bazi podataka SQLite u konfiguraciji. Funkcija db.create_all() neće ponovno kreirati ili ažurirati tablicu baze podataka ako ona već postoji u bazi podataka. Ovo može biti nezgodno kada se modeli modificiraju i promjene moraju biti primijenjene na postojeću bazu podataka. Rješenje grube sile za ažuriranje postojećih tablica baze podataka na drugu shemu je da prvo uklonite stare tablice:

```
>>> db.drop_all() >>>
db.create_all()
```

Nažalost, ova metoda ima neželjenu nuspojavu uništavanja svih podataka u staroj bazi podataka. Bolje rješenje problema ažuriranja baza podataka predstavljeno je pri kraju poglavlja.

Umetanje redova

Sljedeći primjer kreira nekoliko uloga i korisnika:

```
>>> iz hello import Uloga, korisnik >>>
admin_role = Uloga(name='Admin') >>>
mod_role = Uloga(name='Moderator') >>>
user_role = Uloga(name='Korisnik') >>>
user_john = Korisnik(username='john', uloga=admin_role) >>>
user_susan = Korisnik(username='susan', role=user_role) >>>
user_david = Korisnik(username='david', role=user_role)
```

Konstruktori za modele prihvataju početne vrijednosti za atribute modela kao argumente ključne riječi. Imajte na umu da se atribut role može koristiti, iako to nije prava kolona baze podataka, već reprezentacija visokog nivoa odnosa jedan-prema-više. Id _

atribut ovih novih objekata nije eksplisitno postavljen: primarnim ključevima u mnogim bazama podataka upravlja sama baza podataka. Objekti za sada postoje samo na Python strani; još nisu upisani u bazu podataka. Zbog toga njihove id vrijednosti još nisu dodijeljene:

```
>>> print(admin_role.id)
Nema
>>> print(mod_role.id)
Ništa
>>> print(user_role.id)
Nema
```

Promjene u bazi podataka se upravljaju kroz sesiju baze podataka, koju Flask SQLAlchemy pruža kao db.session. Da biste pripremili objekte za pisanje u bazu podataka, oni se moraju dodati u sesiju:

```
>>> db.session.add(admin_role) >>>
db.session.add(mod_role) >>>
db.session.add(user_role) >>>
db.session.add(user_john) >>>
db.session.add(user_susan) >>>
db.session.add(user_david)
```

Ili, sažetije:

```
>>> db.session.add_all([admin_role, mod_role, user_role, user_john,
...                      user_susan, user_david])
```

Da biste upisali objekte u bazu podataka, sesija mora biti urezana pozivanjem njene metode commit() :

```
>>> db.session.commit()
```

Ponovo provjerite id atributne nakon što ste predali podatke da vidite da jesu

sada postavljeno:

```
>>> print(admin_role.id) 1
>>> print(mod_role.id) 2
```

```
>>> print(user_role.id) 3
```



Sesija baze podataka db.session nije povezana s objektom sesije Flask o kojem se govori u poglavljiju 4. Sesije baze podataka se također nazivaju transakcijama.

Sesije baze podataka su izuzetno korisne u održavanju konzistentnosti baze podataka. Operacija urezivanja zapisuje sve objekte koji su dodani sesiji atomski. Ako an

greška se javlja dok se sesija upisuje, cijela sesija se odbacuje. Ako uvijek urezujete povezane promjene zajedno u sesiji, zajamčeno ćete izbjegći nedosljednosti baze podataka zbog djelomičnih ažuriranja.



Sesija baze podataka se također može vratiti.

Ako se pozove db.session.rollback() , svi objekti koji su dodani sesiji baze podataka vraćaju se u stanje koje imaju u bazi podataka.

Modificiranje redova

Add() metoda sesije baze podataka se također može koristiti za ažuriranje modela.

Nastavljajući u istoj sesiji ljske, sljedeći primjer preimenuje ulogu „Admin“ u „Administrator“:

```
>>> admin_role.name = 'Administrator' >>>
db.session.add(admin_role) >>>
db.session.commit()
```

Brisanje redova

Sesija baze podataka takođe ima metodu delete() . Sljedeći primjer briše ulogu "Moderator" iz baze podataka:

```
>>> db.session.delete(mod_role) >>>
db.session.commit()
```

Imajte na umu da se brisanja, poput umetanja i ažuriranja, izvršavaju samo kada je sesija baze podataka predana.

Upitni redovi Flask-

SQLAlchemy čini objekt upita dostupnim u svakoj klasi modela. Najosnovniji upit za model pokreće se metodom all() , koja vraća cijeli sadržaj odgovarajuće tablice:

```
>>> Role.query.all()
[<Uloga 'Administrator'>, <Uloga 'Korisnik'>]
>>> User.query.all()
[<Korisnik 'john'>, <Korisnik 'susan'>, <Korisnik 'david'>]
```

Objekt upita može se konfigurirati za izdavanje specifičnijih pretraživanja baze podataka korištenjem Itera. Sljedeći primjer pronađi sve korisnike kojima je dodijeljena uloga "Korisnik" :

```
>>> User.query.filter_by(role=user_role).all()
[<Korisnik 'susan'>, <Korisnik 'david'>]
```

Također je moguće provjeriti izvorni SQL upit koji SQLAlchemy generira za dati upit pretvaranjem objekta upita u niz:

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS user_id, users.username AS user_username, users.role_id
AS user_role_id \nFROM korisnika \nGDJE :param_1 = users.role_id'
```

Ako izadete iz sesije ljske, objekti kreirani u prethodnom primjeru prestat će postojati kao Python objekti, ali će nastaviti postojati kao redovi u svojim odgovarajućim tablicama baze podataka. Ako tada započnete potpuno novu sesiju ljske, morate ponovo kreirati Python objekte iz njihovih redova baze podataka. Sljedeći primjer izdaje upit koji učitava korisničku ulogu s imenom "Korisnik":

```
>>> user_role = Role.query.filter_by(name='User').first()
```

Obratite pažnju na to kako je u ovom slučaju upit izdat sa metodom first() umjesto all(). Dok all() vraća sve rezultate upita kao listu, first() vraća samo prvi rezultat ili Ništa ako nema rezultata, tako da je zgodna metoda za upotrebu za upite za koje se zna da vraćaju jedan rezultat na većina.

Filteri kao što je filter_by() se pozivaju na objektu upita i vraćaju novi pročišćeni upit. Više filtera se može pozvati u nizu dok se upit ne konfiguriše po potrebi.

Tabela 5-5 prikazuje neke od najčešćih filtera dostupnih upitima. Kompletna lista je u [SQLAlchemy dokumentaciji](#).

Tabela 5-5. Uobičajeni SQLAlchemy upiti Iters

Opcija	Opis
filter()	Vraća novi upit koji dodaje dodatni filter originalnom upitu
filter_by()	Vraća novi upit koji dodaje dodatni filter jednakosti originalnom upitu
limit()	Vraća novi upit koji ograničava broj rezultata originalnog upita na dati broj
offset()	Vraća novi upit koji primjenjuje pomak na listu rezultata originalnog upita
order_by()	Vraća novi upit koji sortira rezultate originalnog upita prema datim kriterijima
group_by()	Vraća novi upit koji grupiše rezultate originalnog upita prema datim kriterijima

Nakon što su željeni filteri primjenjeni na upit, poziv all() će uzrokovati izvršenje upita i vratiti rezultate kao listu—ali postoje i drugi načini da se pokrene izvršenje upita osim all().

Tabela 5-6 prikazuje druge metode izvršenja upita.

Tabela 5-6. Najčešći izvršitelji SQLAlchemy upita

Opcija	Opis
sve()	Vraća sve rezultate upita kao listu
prvi()	Vraća prvi rezultat upita ili Ništa ako nema rezultata
first_or_404()	Vraća prvi rezultat upita ili prekida zahtjev i šalje grešku 404 kao odgovor ako nema rezultata
dobiti()	Vraća red koji odgovara datom primarnom ključu ili Ništa ako se ne pronađe odgovarajući red
get_or_404()	Vraća red koji odgovara datom primarnom ključu ili, ako ključ nije pronađen, prekida zahtjev i šalje grešku 404 kao odgovor
count()	Vraća broj rezultata upita
paginirano()	Vraća objekt Paginacije koji sadrži navedeni raspon rezultata

Relacije funkcionišu slično kao i upiti. Sljedeći primjer ispituje odnos jedan prema mnogo između uloga i korisnika s oba kraja:

```
>>> korisnici = korisnička_uloga.korisnici
>>> korisnici
[<Korisnik 'susan'>, <Korisnik 'david'>]
>>> korisnici[0].role <Uloga 'Korisnik'>
```

Upit user_role.users ovdje ima mali problem. Implicitni upit koji se pokreće kada se izraz user_role.users izda interna poziva all() da vrati listu korisnika. Budući da je objekt upita skriven, nije ga moguće precizirati dodatnim filterima upita. U ovom konkretnom primjeru, možda je bilo korisno zatražiti da se lista korisnika vrati po abecednom redu. U [primjeru 5-4](#), konfiguracija odnosa je modificirana s argumentom lazy='dynamic' kako bi se zatražilo da se upit ne izvrši automatski.

Primjer 5-4. hello.py: dinamički odnosi baze podataka

```
class Uloga(db.Model): #
    korisnici =
        db.relationship('User', backref='role', lazy='dynamic') #
        ...
...
```

Sa odnosom konfiguiranim na ovaj način, user_role.users vraća upit koji još nije izvršen, pa mu se mogu dodati filteri:

```
>>> user_role.users.order_by(User.username).all()
[<Korisnik 'david'>, <Korisnik 'susan'>]
>>> user_role.users.count() 2
```

Upotreba baze podataka u funkcijama prikaza

Operacije baze podataka opisane u prethodnim odjeljcima mogu se koristiti direktno unutar funkcija pogleda.

Primjer 5-5 prikazuje novu verziju rute početne stranice koja bilježi imena koja su korisnici unijeli u bazu podataka.

Primjer 5-5. hello.py: upotreba baze podataka u funkcijama prikaza

```
@app.route('/', methods=['GET', 'POST']) def
index(): form = NameForm() if
form.validate_on_submit():

    user = User.query.filter_by(username=form.name.data).first() ako je korisnik
    Ništa: user = User(username=form.name.data) db.session.add(user)
        db.session.commit() session['poznat'] = Netačno drugačije:
        session['poznat'] = Tačna sesija['name'] = form.name.data
        form.name.data =
        ""

    vrati preusmjeravanje(url_for('index'))
    return render_template('index.html',
        form=form, name=session.get('name'),
        known=session.get('known', False))
```

U ovoj modificiranoj verziji aplikacije, svaki put kada se podnese ime, aplikacija ga provjerava u bazi podataka pomoću filtera upita `filter_by()`. Poznata varijabla se upisuje u korisničku sesiju tako da se nakon preusmjeravanja informacije mogu poslati u šablon, gdje se koriste za prilagođavanje pozdrava. Imajte na umu da da bi aplikacija radila, tabele baze podataka moraju biti kreirane u Python ljudscim kao što je ranije prikazano.

Nova verzija pridruženog predloška prikazana je u **primjeru 5-6**. Ovaj predložak koristi poznati argument za dodavanje drugog reda u pozdrav koji se razlikuje za poznate i nove korisnike.

Primjer 5-6. templates/index.html: prilagođeni pozdrav u šablonu

```
{% proširuje "base.html" %} {%
import "bootstrap/wtf.html" kao wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Zdravo, {{ name }}!{{ else }}Stranac<% endif %!</h1> {% if not
known %}
```

```
<p>Drago mi je što smo se upoznali !
</p> {%- else %} <p>Drago mi je da se
ponovo vidimo!</p> {%- endif %} </div>
{{ wtf.quick_form(form) }} {%- endblock %}
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 5b da provjerite ovu verziju aplikacije.

Integracija sa Python Shell-om

Uvoz instance baze podataka i modela svaki put kada se pokrene sesija ljudske je naporan posao. Da biste izbjegli stalno ponavljanje ovih koraka, naredba flask shell može se konfigurirati da automatski uvozi ove objekte.

Da biste dodali objekte na listu uvoza, procesor konteksta ljudske mora biti kreiran i registrovan u dekoratoru app.shell_context_processor . To je prikazano u [primjeru 5-7.](#)

Primjer 5-7. hello.py: dodavanje konteksta ljudske

```
@app.shell_context_processor def
make_shell_context(): vrati dict(db=db,
Korisnik=Korisnik, Uloga=Uloga)
```

Funkcija procesora konteksta ljudske vraća rječnik koji uključuje instancu baze podataka i modele. Naredba ljudske flask će automatski uvesti ove stavke u ljudsku, pored aplikacije, koja se uvozi po defaultu:

```
$ flask shell
>>> app
<Boca 'zdravo'> >>>
db
<SQLAlchemy engine='sqlite:///home/flask/flasky/data.sqlite'> >>> Korisnik <klasa
'hello.User'>
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 5c da provjerite ovu verziju aplikacije.

Migracije baze podataka sa Flask-Migrate

Kako napredujete u razvoju aplikacije, otkrit ćete da se vaši modeli baze podataka moraju promijeniti, a kada se to dogodi i baza podataka mora biti ažurirana. Flask-SQLAlchemy kreira tabele baze podataka iz modela samo kada oni već ne postoje, tako da je jedini način da se ažuriraju tabele tako da se prvo unište stare tabele—ali, naravno, to dovodi do gubitka svih podataka u bazi podataka.

Bolje rješenje je korištenje okvira za migraciju baze podataka. Na isti način na koji alati za kontrolu verzija izvornog koda prate promjene u datotekama izvornog koda, okvir za migraciju baze podataka prati promjene u šemi baze podataka, omogućavajući primjenu inkrementalnih promjena.

Programer SQLAlchemy napisao je okvir za migraciju pod nazivom [Alembic](#), ali umjesto da direktno koriste Alembic, Flask aplikacije mogu koristiti [Flask-Migrate](#) proširenje, lagani Alembic omot koji ga integriše sa komandom flask .

Kreiranje migracionog spremišta

Za početak, Flask-Migrate mora biti instaliran u virtuelnom okruženju:

```
(venv) $ pip install flask-migrate
```

Primjer 5-8 pokazuje kako se ekstenzija inicijalizira.

Primjer 5-8. hello.py: Inicijalizacija Flask-Migrate

```
from flask_migrate import Migrate

# ...

migracija = Migracija(aplikacija, db)
```

Da bi izložio komande migracije baze podataka, Flask-Migrate dodaje flask db komandu sa nekoliko podnaredbi. Kada radite na novom projektu, možete dodati podršku za migracije baze podataka pomoću podnaredbe init :

```
(venv) $ flask db init
Kreiranje direktorija /home/flask/flasky/migrations...gotovo Kreiranje
direktorija /home/flask/flasky/migrations/versions...gotovo Generiranje /home/
flask/flasky/migrations/alembic.ini ...gotovo Generisanje /home/flask/flasky/
migrations/env.py...done Generisanje /home/flask/flasky/migrations/
env.pyc...done Generisanje /home/flask/flasky/migrations/README..gotovo
Generisanje /home/flask/flasky/migrations/script.py.mako...done Molimo uredite
postavke konfiguracije/veze/logiranja u '/home/flask/flasky/migrations/
alembic.ini' prije nego nastavite.
```

Ova naredba kreira direktorij migracije, gdje će biti pohranjene sve skripte za migraciju. Ako pratite primjer projekta koristeći git checkout, ne morate da radite ovaj korak, jer je spremište za migraciju već uključeno u GitHub spremište.



Datoteke u spremištu za migraciju baze podataka uvijek se moraju dodati u kontrolu verzija zajedno sa ostatkom aplikacije.

Kreiranje skripte za migraciju U

Alembicu, migracija baze podataka je predstavljena skriptom za migraciju. Ova skripta ima dvije funkcije koje se zovu upgrade() i downgrade(). Funkcija upgrade() primjenjuje promjene baze podataka koje su dio migracije, a downgrade() funkcija ih uklanja. Ova mogućnost dodavanja i uklanjanja promena znači da Alembic može rekonfigurisati bazu podataka u bilo kojoj tački u istoriji promena.

Alembičke migracije se mogu kreirati ručno ili automatski koristeći naredbe revizije i migracije . Ručna migracija kreira skriptu skeleta migracije s praznim funkcijama upgrade() i downgrade() koje treba implementirati od strane programera koristeći direktive izložene Alembic-ovom objektu Operations .

Automatska migracija pokušava generirati kod za funkcije upgrade() i downgrade() tražeći razlike između definicija modela i trenutnog stanja baze podataka.



Automatske migracije nisu uvijek tačne i mogu propustiti neke detalje koji su dvomisleni. Na primjer, ako se kolona preimenuje, automatski generirana migracija može pokazati da je dotična kolona izbrisana i da je dodana nova kolona s novim imenom. Ako ostavite migraciju kakva jeste, dovest će do gubitka podataka u ovoj koloni! Iz tog razloga, automatski generisane skripte migracije treba uvek pregledati i ručno ispraviti ako imaju bilo kakve netačnosti.

Da biste izvršili promjene u šemi vaše baze podataka pomoću Flask-Migrate, potrebno je slijediti sljedeću proceduru:

1. Napravite potrebne promjene u klasama modela.
2. Kreirajte skriptu za automatsku migraciju s naredbom flask db migrate .
3. Pregledajte generiranu skriptu i prilagodite je tako da tačno predstavlja promjene koje su napravljene na modelima.

4. Dodajte skriptu migracije u izvornu kontrolu.
5. Primijenite migraciju na bazu podataka s naredbom flask db upgrade .

Podnaredba flask db migrate kreira automatsku skriptu za migraciju:

```
(venv) $ flask db migrirati -m "početna migracija"
INFO [alembic.migration] Kontekst impl SQLiteImpl.
INFO [alembic.migration] Pretpostavlja netransakcioni DDL.
INFO [alembic.autogenerate] Otkrivena dodana tabela 'uloga'
INFO [alembic.autogenerate] Otkrivena dodana tabela 'korisnici'
INFO [alembic.autogenerate.compare] Otkriven dodani indeks
'ix_users_username' na '['username']'
Generiranje /home/flask/flasky/migrations/versions/1bc
594146bb5_initial_migration.py...gotovo
```

Ako slijedite upute git checkout -a da biste postepeno ažurirali primjer aplikacije, ne morate izdavati naredbe za migraciju , jer su skripte za migraciju već ugrađene u oznake Git spremišta.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 5d da provjerite ovu verziju aplikacije. Imajte na umu da ne morate da generišete migraciono spremište i skripte za migraciju za ovu aplikaciju jer su one uključene u GitHub spremište.

Nadogradnja baze podataka

Nakon što je skripta za migraciju pregledana i prihvaćena, može se primjeniti na bazu podataka korištenjem flask db naredbe za nadogradnju:

```
(venv) $ flask db nadogradnja
INFO [alembic.migration] Kontekst impl SQLiteImpl.
INFO [alembic.migration] Pretpostavlja netransakcioni DDL.
INFO [alembic.migration] Pokretanje nadogradnje Ništa -> 1bc594146bb5, početna migracija
```

Za prvu migraciju, ovo je efektivno ekvivalentno pozivanju db.create_all(), ali u uzastopnim migracijama komanda flask db upgrade primjenjuje ažuriranja na tabele bez utjecaja na njihov sadržaj.



Ako ste radili s aplikacijom u prethodnim fazama, već imate datoteku baze podataka koja je ranije kreirana s funkcijom db.create_all(). U ovom stanju, flask db nadogradnja neće uspjeti jer će pokušati kreirati tablice baze podataka koje već postoje. Jednostavan način za rješavanje ovog problema je da izbrišete datoteku baze podataka data.sqlite i zatim pokrenete flask db upgrade da generišete novu bazu podataka kroz okvir migracije.

Druga opcija je da preskočite nadogradnju flask db i umjesto toga označite postojeću bazu podataka kao nadograđenu pomoću naredbe flask db stamp .

Dodavanje više migracija

Dok radite na vlastitim projektima, otkrit ćete da morate vrlo često mijenjati modele baze podataka. Kada upravljate bazom podataka putem okvira za migraciju, sve promjene moraju biti definirane u skriptama migracije, jer sve što nije praćeno u migraciji neće biti ponovljivo. Procedura za uvođenje promjene u bazu podataka je slična onoj koja je urađena za uvođenje prve migracije:

1. Napravite potrebne promjene u modelima baze podataka.
2. Generirajte migraciju s naredbom flask db migrate .
3. Pregledajte generiranu skriptu za migraciju i ispravite je ako ima bilo kakvih netočnosti.
4. Primijenite promjene na bazu podataka s naredbom flask db upgrade .

Dok radite na određenoj funkciji, možda ćete otkriti da trebate napraviti nekoliko promjena u modelima baze podataka prije nego što ih dobijete onako kako želite. Ako vaša posljednja migracija još nije bila posvećena kontroli izvora, možete se odlučiti da je proširite kako biste uključili nove promjene dok ih pravite, a to će vas spasiti od puno malih skripti za migraciju koje su same po sebi besmislene. Procedura za proširenje posljednje migracijske skripte je sljedeća:

1. Uklonite posljednju migraciju iz baze podataka pomoću naredbe flask db downgrade (imajte na umu da to može uzrokovati gubitak nekih podataka).
2. Izbrišite posljednju skriptu za migraciju, koja je sada bez roditelja.
3. Generirajte novu migraciju baze podataka pomoću naredbe flask db migrate , koja će sada uključiti promjene u skripti migracije koju ste upravo uklonili, plus sve druge promjene koje ste napravili na modelima.
4. Pregledajte i primijenite skriptu migracije kao što je prethodno opisano.



Pogledajte [dokumentaciju](#) Flask-Migrate da saznete o drugim podnaredbama koje se odnose na migracije baze podataka.

Tema dizajna i upotrebe baze podataka je veoma važna; napisane su čitave knjige na ovu temu. Trebali biste ovo poglavlje smatrati pregledom; naprednije teme će se raspravljati u kasnijim poglavljima. Sljedeće poglavlje je posvećeno slanju e-pošte.

POGLAVLJE 6

Email

Mnoge vrste aplikacija moraju obavijestiti korisnike kada se dogode određeni događaji, a uobičajeni način komunikacije je e-mail. U ovom poglavlju ćete naučiti kako slati e-poštu iz Flask aplikacije.

Podrška putem e-pošte sa Flask-Mail

Iako se paket smtplib iz standardne biblioteke Python može koristiti za slanje e-pošte unutar Flask aplikacije, proširenje Flask-Mail obavlja smtplib i lijepo ga integrira sa Flaskom. Flask-Mail se instalira sa pip-om:

```
(venv) $ pip install flask-mail
```

Ekstenzija se povezuje sa serverom Simple Mail Transfer Protocol (SMTP) i proslijeđuje mu e-poštu radi isporuke. Ako nije data konfiguracija, Flask-Mail se povezuje na localhost na portu 25 i šalje e-poštu bez autentifikacije. **Tabela 6-1** prikazuje listu konfiguracionih ključeva koji se mogu koristiti za konfiguriranje SMTP servera.

Tabela 6-1. Ključevi za konfiguraciju SMTP servera Flask-Mail

Ključ	Zadani opis
MAIL_SERVER	localhost Ime hosta ili IP adresa servera e-pošte
MAIL_PORT	25 Port servera e-pošte
MAIL_USE_TLS	False Omogući sigurnost transportnog sloja (TLS).
MAIL_USE_SSL	False Omogući sigurnost sloja sigurnih utičnica (SSL) .
MAIL_USERNAME	Nema Korisničko ime Mail računa
MAIL_PASSWORD	Ništa Lozinka naloga za poštu

Tokom razvoja može biti zgodnije povezati se sa eksternim SMTP serverom. Kao primjer, [primjer 6-1](#) pokazuje kako konfigurirati aplikaciju za slanje e-pošte putem Google Gmail računa.

Primjer 6-1. hello.py: Flask-Mail konfiguracija za Gmail

```
uvezi nas #
...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587 app.config['MAIL_USE_TLS'] =
Tačno app.config['MAIL_USERNAME'] = os.environ.get
('MAIL_USERNAME') app.config['MAIL_PASSWORD'] =
os.environ.get('MAIL_PASSWORD')
```



Nikada nemojte pisati vjerodajnice računa direktno u svoje skripte, posebno ako planirate objaviti svoj rad kao open source. Da biste zaštitili informacije o vašem računu, neka vaša skripta uvozi osjetljive informacije iz varijabli okruženja.



Iz sigurnosnih razloga, Gmail nalozi su konfigurisani tako da zahtijevaju da vanjske aplikacije koriste OAuth2 autentifikaciju za povezivanje sa serverom e-pošte. Nažalost, Pythonova smtplib biblioteka ne podržava ovu metodu provjere autentičnosti. Da bi vaš Gmail nalog prihvatio standardnu SMTP autentifikaciju, idite na [stranicu postavki](#) vašeg Google računa i odaberite "Prijava na Google" na lijevoj traci menija. Na toj stranici pronađite postavku „Dozvoli manje sigurne aplikacije“ i uvjerite se da je omogućena. Ako se omogućavanje ove postavke na vašem ličnom Gmail računu tiče, kreirajte sekundarni račun samo da biste testirali slanje e-pošte.

Flask-Mail se inicijalizira kao što je prikazano u [primjeru 6-2](#).

Primjer 6-2. hello.py: Inicijalizacija Flask-Mail

```
from flask_mail import Mail mail
= Mail(app)
```

Dvije varijable okruženja koje drže korisničko ime i lozinku servera e-pošte moraju biti definirane u okruženju. Ako koristite Linux ili macOS, ove varijable možete postaviti na sljedeći način:

```
(venv) $ export MAIL_USERNAME=<Gmail korisničko ime>
(venv) $ export MAIL_PASSWORD=<Gmail lozinka>
```

Za korisnike Microsoft Windowsa, varijable okruženja su postavljene na sljedeći način:

```
(venv) $ set MAIL_USERNAME=<Gmail korisničko  
ime> (venv) $ set MAIL_PASSWORD=<Gmail lozinka>
```

Slanje e-pošte iz Python Shell -a Da biste testirali

konfiguraciju, možete započeti sesiju ljske i poslati probnu e-poštu (zamjenite you@example.com sa svojom vlastitom adresom e-pošte):

```
(venv) $ flask shell >>> iz  
flask_mail import poruke >>> iz hello  
import mail >>> msg = Poruka('test email',  
sender='you@example.com', recipients=['you@example. com'])  
...  
>>> msg.body = 'Ovo je tijelo običnog teksta' >>>  
msg.html = 'Ovo je <b>HTML</b> tijelo' >>> sa  
app.app_context(): mail.send( poruka)  
...  
...
```

Imajte na umu da funkcija send() Flask-Maila koristi current_app, tako da se mora izvršiti s aktiviranim kontekstom aplikacije.

Integracija e-poruka sa aplikacijom Kako biste izbjegli

potrebe da svaki put ručno kreirate poruke e-pošte, dobra je ideja apstrahovati uobičajene dijelove funkcije slanja e-pošte aplikacije u funkciju. Kao dodatnu prednost, ova funkcija može prikazati tijela e-pošte sa Jinja2 šablona kako bi imala najveću fleksibilnost. Implementacija je prikazana u [primjeru 6-3](#).

Primjer 6-3. hello.py: podrška putem e-pošte

```
iz flask_mail import poruke

app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'  
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin <flasky@example.com>'

def send_email(za, predmet, šablon, **kwargs): msg =  
    Poruka(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + predmet,  
           sender=app.config['FLASKY_MAIL_SENDER'], recipients=[za]) msg.body  
    = render_template(template + '.txt', **kwargs) msg.html = render_template(template +  
    '.html', **kwargs ) mail.send(msg)
```

Funkcija se oslanja na dva konfiguracijska ključa specifična za aplikaciju koja definiraju niz prefiksa za predmet i adresu koja će se koristiti kao pošiljatelj. Funkcija send_email() uzima odredišnu adresu, liniju predmeta, šablon za tijelo e-pošte i listu argumenata ključne riječi. Ime šablona mora biti navedeno

bez ekstenzije, tako da se dvije verzije šablona mogu koristiti za običan tekst i HTML tijela. Argumenti ključne riječi koje je proslijedio pozivatelj daju se render_template() tako da ih predlošci koji generiraju tijelo e-pošte mogu koristiti kao varijable šablona.

Funkcija prikaza index() može se lako proširiti da pošalje e-poruku administratoru kad god se uz obrazac primi novo ime. **Primjer 6-4** pokazuje ovu promjenu.

Primjer 6-4. hello.py: primjer e-pošte

```
...
# app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN') #
...
@app.route('/', methods=['GET', 'POST']) def
index(): form = NameForm() if
form.validate_on_submit():

    user = User.query.filter_by(username=form.name.data).first() ako je
    korisnik Ništa: user = User(username=form.name.data) db.session.add(user)
        session['poznat'] = False if app.config['FLASKY_ADMIN']:
            send_email(app.config['FLASKY_ADMIN'], 'Novi korisnik', 'mail/
            new_user', user=user)

else:
    session['known'] = Tačna
    sesija['name'] = form.name.data
    form.name.data = return
    redirect(url_for('index')) return
render_template('index.html', form=form , name=session.get('name'),
    poznato=session.get('poznato', False))
```

Primalac e-pošte je dat u varijabli okruženja FLASKY_ADMIN koja se učitava u konfiguracionu varijablu istog imena tokom pokretanja. Potrebno je kreirati dvije datoteke šablona za tekstualnu i HTML verziju e-pošte. Ove datoteke se pohranjuju u poddirektoriju pošte unutar šablona kako bi bile odvojene od običnih šablona. Šabloni e-pošte očekuju da korisnik bude dat kao argument šablona, tako da ga poziv send_email() uključuje kao argument ključne riječi.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 6a da provjerite ovu verziju aplikacije.

Pored varijabli okruženja MAIL_USERNAME i MAIL_PASSWORD opisanih ranije, ovoj verziji aplikacije potrebna je varijabla okruženja FLASKY_ADMIN . Za korisnike Linuxa i macOS-a, naredba za postavljanje ove varijable je:

```
(venv) $ export FLASKY_ADMIN=<vaša-e-mail adresa>
```

Za korisnike Microsoft Windows-a, ovo je ekvivalentna naredba:

```
(venv) $ set FLASKY_ADMIN=<vaša-e-mail adresa>
```

Sa ovim postavljenim varijablama okruženja, možete testirati aplikaciju i primati e-poštu svaki put kada unesete novo ime u obrazac.

Slanje asinhronih e-pošta Ako ste

poslali nekoliko probnih e-poruka, vjerovatno ste primijetili da se funkcija mail.send() blokira na nekoliko sekundi dok se e-pošta šalje, zbog čega pretraživač za to vrijeme ne reagira. Kako bi se izbjegla nepotrebna kašnjenja tokom obrade zahtjeva, funkcija slanja e-pošte može se premjestiti u pozadinsku nit. **Primjer 6-5** pokazuje ovu promjenu.

Primjer 6-5. hello.py: podrška za asinhroni email

```
iz threading import Thread

def send_async_email(app, msg):
    app.app_context().mail.send(msg)

def send_email(za, predmet, šablon, **kwargs):
    msg = Poruka(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + predmet,
                sender=app.config['FLASKY_MAIL_SENDER'], recipients=[za])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return thr
```

Ova implementacija naglašava zanimljiv problem. Mnoge Flask ekstenzije rade pod pretpostavkom da postoje aktivni konteksti aplikacije i/ili zahtjeva. Kao što je već spomenuto, funkcija send() Flask-Maila koristi current_app, tako da zahtjeva da kontekst aplikacije bude aktivan. Ali budući da su konteksti povezani s niti, kada se funkcija mail.send() izvrši u drugoj niti, potrebno je da se kontekst aplikacije kreira umjetno koristeći app.app_context(). Instanca aplikacije se proslijeđuje niti kao argument tako da se može kreirati kontekst.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 6b da provjerite ovu verziju aplikacije.

Ako sada pokrenete aplikaciju, primjetit ćete da mnogo bolje reagira, ali imajte na umu da je za aplikacije koje šalju veliku količinu e-pošte prikladnije imati posao posvećen slanju e-pošte nego pokretanje nove niti za svaku e-poštu poslati operaciju. Na primjer, izvršenje funkcije `send_async_email()` može se poslati na [Celery](#) red zadataka.

Ovo poglavlje završava pregled funkcija koje su neophodne za većinu web aplikacija. Sada je problem u tome što skripta `hello.py` počinje da raste i to otežava rad sa njom. U sljedećem poglavlju naučit ćete kako strukturirati veću aplikaciju.

GLAVA 7

Velika aplikaciona struktura

Iako imanje malih web aplikacija pohranjenih u jednoj datoteci skripte može biti vrlo zgodno, ovaj pristup nije dobro skaliran. Kako aplikacija postaje sve složenija, rad s jednom velikom izvornom datotekom postaje problematičan.

Za razliku od većine drugih web okvira, Flask ne nameće posebnu organizaciju za velike projekte; način strukturiranja aplikacije je u potpunosti prepušten programeru. U ovom poglavlju predstavljen je mogući način organiziranja velike aplikacije u paketima i modulima. Ova struktura će se koristiti u preostalim primjerima knjige.

Struktura projekta

Primjer 7-1 prikazuje osnovni izgled aplikacije Flask.

Primjer 7-1. Osnovna višestruka struktura Flask aplikacije

```
|-flasky |-
  app/ |-
    templates/ |-
      static/ |-
        main/
        |-.init_.py |-
        errors.py |-
        forms.py |-
        views.py |-
        __init__.py |-
        email.py |
        -models.py |
        migrations/ |-
    tests/ |-.init_.py |-
    test*.py |-
    venv/ |-
    requirements.txt
    |-config.py |-
  flasky.py
```

Ova struktura ima četiri foldera najvišeg nivoa:

- Flask aplikacija živi unutar paketa koji se naziva aplikacija.
- Fascikla migracija sadrži skripte za migraciju baze podataka, kao i ranije.
- Jedinični testovi su napisani u paketu testova.
- Fascikla venv sadrži Python virtuelno okruženje, kao i ranije.

Tu je i nekoliko novih fajlova:

- Zahtevi.txt navodi zavisnosti paketa tako da je lako regenerisati identično virtuelno okruženje na drugom računaru.
- cong.py pohranjuje postavke konfiguracije.
- flasky.py definiše instancu aplikacije Flask, a uključuje i nekoliko zadataka koji pomozite u upravljanju aplikacijom.

Kako biste u potpunosti razumjeli ovu strukturu, sljedeći odeljci opisuju proces pretvaranja aplikacije hello.py u nju.

Opcije konfiguracije

Aplikacije često trebaju nekoliko konfiguracijskih skupova. Najbolji primjer za to je potreba da se koriste različite baze podataka tokom razvoja, testiranja i proizvodnje kako ne bi ometale jedna drugu.

Umjesto jednostavne konfiguracije poput rječnika app.config koju koristi hello.py, može se koristiti hijerarhija konfiguracijskih klasa. **Primjer 7-2** prikazuje datoteku cong.py, sa svim postavkama uvezenim iz hello.py.

Primjer 7-2. cong.py: konfiguracija aplikacije

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'teško pogoditi niz'
    MAIL_SERVER = os.environ.get('MAIL_SERVER', 'smtp.googlemail.com')
    MAIL_PORT = int(os.environ.get('MAIL_PORT', '587'))
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() or \
        ['tačno', 'uključeno', '1']
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
    FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')
    SQLALCHEMY_TRACK_MODIFICATIONS = Netačno
```

```

@staticmethod
def init_app(app): prolaz

klasa DevelopmentConfig(Config):
    DEBUG = Tačno
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') ili \ 'sqlite:///'
        + os.path.join(basedir, 'data-dev.sqlite')

klasa TestingConfig(Config):
    TESTIRANJE = Tačno
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') ili \
        'sqlite://'

klasa ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') ili \ 'sqlite:///'
        + os.path.join(basedir, 'data.sqlite')

config =
    { 'development': DevelopmentConfig,
      'testing': TestingConfig, 'production':
      ProductionConfig,
      'podrazumevano': DevelopmentConfig
    }

```

Osnovna klasa Config sadrži postavke koje su zajedničke za sve konfiguracije; različite podklase definiraju postavke koje su specifične za konfiguraciju. Po potrebi se mogu dodati dodatne konfiguracije.

Da bi konfiguracija bila fleksibilnija i bezbednija, većina podešavanja se opcionalno može uvesti iz varijabli okruženja. Na primjer, vrijednost SECRET_KEY, zbog njegove osjetljive prirode, može biti postavljena u okruženju, ali je zadana vrijednost data u slučaju da je okruženje ne definira. Obično se ove postavke mogu koristiti sa svojim zadanim vrijednostima tokom razvoja, ali svaka bi trebala imati odgovarajuću vrijednost postavljenu u odgovarajuću varijablu okruženja na proizvodnom serveru. Opcije konfiguracije za server e-pošte su također uvezene iz varijabli okruženja, sa zadanim vrijednostima koje upućuju na Gmail server radi pogodnosti tokom razvoja.



Nikada nemojte pisati lozinke ili druge tajne u konfiguracijsku datoteku koja je predana kontroli izvora.

Varijabli SQLALCHEMY_DATABASE_URI su dodijeljene različite vrijednosti pod svakom od tri konfiguracije. Ovo omogućava aplikaciji da koristi različite baze podataka u svakoj od njih

konfiguraciju. Ovo je veoma važno, jer ne želite da izvođenje jediničnih testova promeni bazu podataka koju koristite za svakodnevni razvoj. Svaka konfiguracija pokušava uvesti URL baze podataka iz varijable okruženja, a kada ona nije dostupna postavlja zadalu baziranu na SQLite-u. Za konfiguraciju testiranja, zadana je baza podataka u memoriji, budući da nema potrebe za pohranjivanjem bilo kakvih podataka izvan probnog pokretanja.

Svaka konfiguracija razvoja i proizvodnje ima skup opcija konfiguracije mail servera. Kao dodatni način da se dozvoli aplikaciji da prilagodi svoju konfiguraciju, klasa Config i njene podklase mogu definirati metodu klase init_app() koja uzima instancu aplikacije kao argument. Za sada osnovna klasa Config implementira praznu metodu init_app().

Na dnu konfiguracijske skripte, različite konfiguracije su registrirane u konfiguracijskom rječniku. Jedna od konfiguracija (u ovom slučaju ona za razvoj) je također registrirana kao zadana.

Paket aplikacija

Paket aplikacije je mjesto gdje žive svi kodovi aplikacije, predlošci i statički fajlovi. Ovdje se zove jednostavno aplikacija, iako joj se po želji može dati naziv specifičan za aplikaciju. Predlošci i statički direktoriji sada su dio paketa aplikacije, tako da se premeštaju unutar aplikacije. Modeli baze podataka i funkcije podrške za e-poštu također se premeštaju unutar ovog paketa, svaki u svom modulu, kao app/models.py i app/email.py.

Korištenje tvornice aplikacija Način

na koji se aplikacija kreira u verziji sa jednim fajlom je vrlo zgodan, ali ima jedan veliki nedostatak. Budući da je aplikacija kreirana u globalnom opsegu, ne postoji način da se promjene konfiguracije primijene dinamički: u vrijeme kada se skripta pokrene, instanca aplikacije je već kreirana, tako da je već prekasno za promjenu konfiguracije. Ovo je posebno važno za jedinične testove jer je ponekad potrebno pokrenuti aplikaciju pod različitim postavkama konfiguracije radi bolje pokrivenosti testom.

Rješenje ovog problema je odgoditi kreiranje aplikacije premeštanjem u fabričku funkciju koja se može eksplisitno pozvati iz skripte. Ovo ne samo da daje skripti vremena za postavljanje konfiguracije, već i mogućnost kreiranja više instanci aplikacije – još jedna stvar koja može biti vrlo korisna tokom testiranja. Tvornička funkcija aplikacije, prikazana u [primjeru 7-3](#), definirana je u paketu aplikacije konstruktor.

Primjer 7-3. app/_init_.py: konstruktor paketa aplikacije

```
from flask import Flask, render_template
flask_bootstrap import Bootstrap from flask_mail
import Mail from flask_moment import Moment
from flask_sqlalchemy import SQLAlchemy from
config import config

bootstrap = Bootstrap() mail
= Mail() moment = Moment()
db = SQLAlchemy()

def create_app(config_name): app
    = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

# priložite rute i prilagođene stranice grešaka ovdje

povratna aplikacija
```

Ovaj konstruktor uvozi većinu ekstenzija Flask-a koje se trenutno koriste, ali pošto ne postoji instanca aplikacije kojom bi ih inicijalizirala, stvara ih neinicijalizirane tako što ne proslijede argumente u njihove konstruktore. Funkcija `create_app()` je tvornica aplikacije, koja kao argument uzima ime konfiguracije za korištenje za aplikaciju. Postavke konfiguracije pohranjene u jednoj od klasa definiranih u `cong.py` mogu se uvesti direktno u aplikaciju korištenjem metode `from_object()` dostupnog u Flask-ovom `app.config` konfiguracijskom objektu. Konfiguracijski objekt se bira po imenu iz konfiguracijskog rječnika. Kada se aplikacija kreira i konfiguriše, ekstenzije se mogu inicijalizirati. Pozivanje `init_app()` na ekstenzijama koje su ranije kreirane dovršava njihovu inicijalizaciju.

Inicijalizacija aplikacije se sada vrši u ovoj tvorničkoj funkciji, koristeći metodu `from_object()` iz konfiguracijskog objekta Flask, koja kao argument uzima jednu od konfiguracijskih klasa definiranih u `cong.py`. Metoda `init_app()` odabrane konfiguracije se također poziva, kako bi se omogućile složenije procedure inicijalizacije.

Tvornička funkcija vraća kreiranu instancu aplikacije, ali imajte na umu da su aplikacije kreirane s fabričkom funkcijom u svom trenutnom stanju nekompletne, jer im nedostaju rute i prilagođeni rukovaoci stranicama grešaka. Ovo je tema sljedećeg odjeljka.

Implementacija funkcionalnosti aplikacije u nacrtu Konverzija u tvornicu aplikacija uvodi komplikaciju za rute. U aplikacijama sa jednom skriptom, instanca aplikacije postoji u globalnom opsegu, tako da se rute mogu lako definirati korištenjem app.route dekoratora. Ali sada kada je aplikacija kreirana u vrijeme izvođenja, dekorator app.route počinje postojati tek nakon što se pozove create_app() , što je prekasno. Prilagođeni rukovaoci stranicama grešaka predstavljaju isti problem, jer su definisani dekoratorom app.errorhandler .

Srećom, Flask nudi bolje rješenje koristeći nacrte. Nacrt je sličan aplikaciji po tome što također može definirati rute i rukovaće greškama. Razlika je u tome što kada su oni definisani u nacrtu, oni su u stanju mirovanja sve dok se nacrt ne registruje u aplikaciji, u kom trenutku oni postaju deo nje. Koristeći nacrt definisan u globalnom opsegu, rute i rukovaoci greškama aplikacije mogu se definisati na skoro isti način kao u aplikaciji sa jednom skriptom.

Kao i aplikacije, nacrti se mogu definirati u jednoj datoteci ili se mogu kreirati na strukturiranoj način s više modula unutar paketa. Kako bi se omogućila najveća fleksibilnost, kreirat će se potpaket unutar paketa aplikacije za smještaj prvog plana aplikacije. **Primjer 7-4** prikazuje konstruktor paketa koji kreira nacrt.

Primjer 7-4. app/main/__init__.py: kreiranje glavnog nacrta

```
iz flask import Blueprint

main = Blueprint('main', __name__)

od . uvoz pogleda, greške
```

Nacrti se kreiraju instanciranjem objekta klase Blueprint. Konstruktor za ovu klasu uzima dva potrebna argumenta: ime nacrta i modul ili paket u kojem se nacrt nalazi. Kao i kod aplikacija, Pythonova varijabla __name__ je u većini slučajeva ispravna vrijednost za drugi argument.

Rute aplikacije su pohranjene u modulu app/main/views.py unutar paketa, a rukovaoci greškama su u app/main/errors.py. Uvoz ovih modula dovodi do toga da se rute i rukovaoci greškama pridruže nacrtu. Važno je napomenuti da se moduli uvoze na dnu app/main/__init__.py skripte kako bi se izbjegle greške zbog kružnih ovisnosti. U ovom konkretnom primjeru problem je u tome što će app/main/views.py i app/main/errors.py zauzvrat uvesti glavni objekt nacrta, tako da uvoz neće uspjeti osim ako se kružna referenca ne dogodi nakon što je main definiran .



The `from . import <some-module>` sintaksa se koristi u Pythonu za predstavljanje relativnog putanja. Trenutno je javi Uskoro ćete vidjeti još jedan vrlo koristan relativni uvoz koji koristi obrazac iz `.. import <some-module>`.

Nacrt je registrovan u aplikaciji unutar tvorničke funkcije `create_app()`, kao što je prikazano u [primjeru 7-5](#).

[Primjer 7-5.](#) `app/__init__.py`: registracija glavnog plana

```
def create_app(config_name):
    # ...

    iz .main import main kao main_blueprint
    app.register_blueprint(main_blueprint)

    povratna aplikacija
```

[Primjer 7-6](#) prikazuje rukovaće greškama.

[Primjer 7-6.](#) `app/main/errors.py`: rukovaoci grešaka u glavnom nacrtu

```
iz flask import render_template iz . import
main

@main.app_errorhandler(404) def
page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500) def
internal_server_error(e): return
    render_template('500.html'), 500
```

Razlika kod pisanja rukovaoca greškama unutar nacrta je u tome što ako se koristi dekorator za obradu grešaka , rukovalac će biti pozvan samo za greške koje potiču na rutama definisanim nacrtom. Da biste instalirali rukovaće greškama u cijeloj aplikaciji, umjesto toga se mora koristiti dekorator `app_errorhandler` .

[Primjer 7-7](#) prikazuje rutu aplikacije ažuriranu da bude u nacrtu.

Primjer 7-7. app/main/views.py: rute aplikacije u glavnom nacrtu

```
from datetime import datetime
from flask import render_template, session, redirect, url_for from . import main
from .forms import NameForm from .. import db from ..models import User

# ...
# ...
```

`@main.route('/', methods=['GET', 'POST']) def index(): form = NameForm() if form.validate_on_submit():`

```
# ...
return redirect(url_for('.index'))
return render_template('index.html',
form=form, name=session.get('name'),
known=session.get('known', False),
current_time=datetime.utcnow())
```

Postoje dvije glavne razlike kada se piše funkcija prikaza unutar nacrta. Prvo, kao što je ranije urađeno za rukovače greškama, dekorator rute dolazi iz nacrtu, tako da se main.route koristi umjesto app.route. Druga razlika je u korištenju funkcije url_for(). Kao što se možda sjećate, prvi argument ove funkcije je naziv krajnje točke rute, koji je za rute zasnovane na aplikaciji zadano ime funkcije prikaza. Na primjer, u aplikaciji sa jednom skriptom URL za funkciju prikaza index() može se dobiti pomoću url_for('index').

Razlika sa nacrtima je u tome što Flask primjenjuje imenski prostor na sve krajnje točke definirane u nacrtu, tako da višestruki nacrti mogu definirati funkcije pogleda sa istim imenima krajnjih tačaka bez kolizija. Imenski prostor je ime nacrtu (prvi argument konstruktora Blueprint) i odvojen je od imena krajnje tačke tačkom. Funkcija prikaza index() se zatim registruje sa imenom krajnje tačke main.index i njen URL se može dobiti sa url_for('main.index').

Funkcija url_for() takođe podržava kraći format za krajnje tačke u nacrtima u kojima je ime nacrtu izostavljeno, kao što je url_for('.index'). Sa ovom notacijom, naziv nacrtu za trenutni zahtjev se koristi za dovršavanje naziva krajnje točke. Ovo efektivno znači da preusmjeravanja unutar istog nacrtu mogu koristiti kraći oblik, dok preusmjeravanja preko nacrtu moraju koristiti potpuno kvalificirano ime krajnje točke koje uključuje ime nacrtu.

Da biste dovršili promjene u paketu aplikacije, objekti obrasca se također pohranjuju unutar nacrtu u modulu app/main/forms.py.

Application Script

Modul flasky.py u direktoriju najviše razine je mjesto gdje je definirana instanca aplikacije. Ova skripta je prikazana u [primjeru 7-8.](#)

Primjer 7-8. flasky.py: glavna skripta

```
import os
iz aplikacije import create_app, db iz
app.models import User, uloga iz
flask_migrate import Migrate

app = create_app(os.getenv('FLASK_CONFIG') ili 'default') migrate =
Migrate(app, db)

@app.shell_context_processor def
make_shell_context(): vrati
dict(db=db, Korisnik=Korisnik, Uloga=Uloga)
```

Skripta počinje kreiranjem aplikacije. Konfiguracija se uzima iz varijable okruženja FLASK_CONFIG ako je definirana, ili se u suprotnom koristi zadana konfiguracija. Flask-Migrate i prilagođeni kontekst za Python shell se tada inicijaliziraju.

Budući da se glavna skripta aplikacije promjenila iz hello.py u flasky.py, varijablu okruženja FLASK_APP treba ažurirati u skladu s tim kako bi komanda flask mogla locirati instancu aplikacije. Također je korisno omogućiti Flask-ov način otklanjanja grešaka postavljanjem FLASK_DEBUG=1. Za Linux i macOS, sve se to radi na sljedeći način:

```
(venv) $ export FLASK_APP=flasky.py (venv)
$ export FLASK_DEBUG=1
```

I za Microsoft Windows:

```
(venv) $ set FLASK_APP=flasky.py (venv)
$ set FLASK_DEBUG=1
```

Requirements File

Dobra je praksa da aplikacije uključe zahtjeve.txt datoteku koja bilježi sve ovisnosti paketa, s tačnim brojevima verzija. Ovo je važno u slučaju da virtualno okruženje treba regenerisati na drugom računaru, kao što je mašina na kojoj će aplikacija biti raspoređena za proizvodnu upotrebu. Ovu datoteku može automatski generirati pip sa sljedećom naredbom:

```
(venv) $ pip zamrzavanje >requirements.txt
```

Dobra je ideja osježiti ovu datoteku kad god se paket instalira ili nadograđi. Ovdje je prikazan primjer datoteke sa zahtjevima:

```

alembic==0.9.3
blinker==1.4
klik==6.7
dominirati==2.3.1
Flask==0.12.2 Flask-
Bootstrap==3.3.7.1 Flask-
Mail==0.9.1 Flask-
Migrate==2.0.4 Flask-
Moment==0.5.1 Flask-
SQLAlchemy==2.2 Flask-
WTForms==0.14.2 itsdangerous==0.24
Jinja2==2.9.6 Mako==1.0.7
MarkupSafe==1.0 python-
dateutil==2.6.1 python-
editor==1.0.3

šest==1.10.0
SQLAlchemy==1.1.11
visitor==0.1.3
Alat==0.12.2
WTForms==2.1

```

Kada trebate napraviti savršenu repliku virtualnog okruženja, možete kreirati novo virtualno okruženje i na njemu pokrenuti sljedeću naredbu:

```
(venv) $ pip install -r zahtjevi.txt
```

Brojevi verzija u primjeru datoteke requirements.txt će vjerovatno biti zastarjeli do trenutka kada ovo pročitate. Možete pokušati koristiti novija izdanja paketa, ako želite. Ako imate bilo kakvih problema, uvijek se možete vratiti na verzije navedene ovdje, jer se zna da su one kompatibilne s aplikacijom.

Jedinični testovi

Ova aplikacija je vrlo mala, tako da još uvijek nema puno toga za testiranje. Ali kao primjer, mogu se definirati dva jednostavna testa, kao što je prikazano u [primjeru 7-9](#).

Primjer 7-9. tests/test Basics.py: jedinični testovi

```

import unittest
from flask import current_app
from . import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testiranje')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

```

```

def tearDown(self):
    db.session.remove()
    db.drop_all()
    self.app_context.pop()

def test_app_exists(self):
    self.assertFalse(current_app is None)

def test_app_is_testing(self):
    self.assertTrue(current_app.config['TESTING'])

```

Testovi su napisani pomoću standardnog paketa unittest iz Python standardne biblioteke. Metode setUp() i tearDown() klase test case se pokreću prije i nakon svakog testa, a izvršavaju se sve metode koje imaju ime koje počinje sa test_ kao testovi.



Ako želite da saznate više o pisanju jediničnih testova sa Pythonovim unittest paketom, pročitajte [zvaničnu dokumentaciju](#).

Metoda setUp() pokušava stvoriti okruženje za test koji je blisko okruženju aplikacije koja radi. Prvo kreira aplikaciju konfiguriranu za testiranje i aktivira njen kontekst. Ovaj korak osigurava da testovi imaju pristup current_app, kao što to rade obični zahtjevi. Zatim kreira potpuno novu bazu podataka za testove koristeći metodu create_all() Flask SQLAlchemy . Baza podataka i kontekst aplikacije uklanjanju se metodom tearDown() .

Prvi test osigurava da instanca aplikacije postoji. Drugi test osigurava da aplikacija radi pod konfiguracijom za testiranje. Da bi direktorij testova bio ispravan paket, potrebno je dodati modul tests/init.py, ali to može biti prazna datoteka, jer paket unittest skenira sve module kako bi otkrio testove.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 7a da provjerite konvertovanu verziju aplikacije. Da biste bili sigurni da imate instalirane sve ovisnosti, također pokrenite pip install -r requirements.txt.

Za pokretanje jediničnih testova, prilagođena komanda se može dodati skripti flasky.py. [Primjer 7-10](#) pokazuje kako dodati test naredbu.

Primjer 7-10. flasky.py: naredba pokretača testa jedinica

```
@app.cli.command()
def test(): """Pokreni
    testove jedinica."""
    import
    unittest tests =
    unittest.TestLoader().discover('testovi')
    unittest.TextTestRunner(verbosity=2).run (testovi)
```

App.cli.command dekorator olakšava implementaciju prilagođenih komandi. Ime ukrašene funkcije koristi se kao ime naredbe, a docstring funkcije se prikazuje u porukama pomoći. Implementacija funkcije test() poziva pokretač testa iz paketa unittest .

Jedinični testovi se mogu izvršiti na sljedeći način:

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ...
test_app_is_testing (test_basics.BasicsTestCase) ...

=====
Obavljeni 2 testa za 0,001s

=====
```

Podešavanje baze podataka

Restrukturirana aplikacija koristi drugačiju bazu podataka od verzije sa jednom skriptom. URL baze podataka je uzet iz varijable okruženja kao prvi izbor, sa zadanim SQLite bazom podataka kao alternativom. Varijable okruženja i imena datoteka SQLite baze podataka razlikuju se za svaku od tri konfiguracije. Na primjer, u razvojnoj konfiguraciji URL se dobiva iz varijable okruženja DEV_DATABASE_URL, a ako to nije definirano onda se koristi SQLite baza podataka s imenom data-dev.sqlite.

Bez obzira na izvor URL-a baze podataka, tabele baze podataka moraju biti kreirane za novu bazu podataka. Kada radite sa Flask-Migrate da biste pratili migracije, tabele baze podataka mogu se kreirati ili nadograditi na najnoviju reviziju pomoću jedne naredbe:

```
(venv) $ flask db nadogradnja
```

Pokretanje aplikacije

Refaktoriranje je sada završeno i aplikacija se može pokrenuti. Uvjerite se da ste ažurirali varijablu okruženja FLASK_APP kao što je navedeno u "["Skripta aplikacije" na stranici 93](#), a zatim pokrenite aplikaciju kao i obično:

```
(venv) $ flask run
```

Postavljanje varijabli okruženja FLASK_APP i FLASK_DEBUG svaki put kada se pokrene nova sesija komandne linije može biti zamorno, tako da biste trebali konfigurirati svoj sistem tako da ove varijable budu postavljene po defaultu. Ako koristite bash, možete ih dodati u svoj `~/.bashrc` fajl.

Vjerovali ili ne, došli ste do kraja prvog dijela. Sada ste naučili o osnovnim elementima potrebnim za pravljenje web aplikacije sa Flaskom, ali vjerovatno niste sigurni kako se svi ovi dijelovi uklapaju u pravu aplikaciju. Cilj II dijela je pomoći u tome tako što će vas provesti kroz razvoj kompletne aplikacije.

DIO II

Primjer: Društveni
Aplikacija za bloganje

Autentifikacija korisnika

Većina aplikacija treba da prati ko su njihovi korisnici. Kada se korisnici povežu s aplikacijom, oni se autentikuju s njom, proces kojim obznanjuju svoj identitet. Kada aplikacija zna ko je korisnik, može ponuditi prilagođeno iskustvo.

Najčešće korištena metoda autentifikacije zahtjeva od korisnika da dostave dio identifikacije, a to je ili njihova adresa e-pošte ili korisničko ime, i tajnu koja je samo njima poznata, a koja se zove lozinka. U ovom poglavlju kreiran je kompletan sistem autentikacije za Flasky.

Proširenja za provjeru autentičnosti za Flask

Postoji mnogo odličnih Python paketa za autentifikaciju, ali nijedan od njih ne radi sve. Rješenje za provjeru autentičnosti korisnika predstavljeno u ovom poglavlju koristi nekoliko paketa i pruža ljestvico koje čini da oni dobro funkcioniraju zajedno. Ovo je lista paketa koji će se koristiti i za šta se koriste:

- Flask-Login: Upravljanje korisničkim sesijama za prijavljene korisnike
- Werkzeug: heširanje i verifikacija lozinke
- opasno: Generisanje i verifikacija kriptografski bezbednog tokena

Pored paketa specifičnih za autentifikaciju, koristit će se sljedeća proširenja opće namjene:

- Flask-Mail: Slanje emailova vezanih za autentifikaciju
- Flask-Bootstrap: HTML šabloni
- Flask-WTF: Web obrasci

Sigurnost lozinke

Sigurnost korisničkih informacija pohranjenih u bazama podataka često se zanemaruje tokom dizajna web aplikacija. Ako napadač uspije provaliti na vaš server i pristupiti vašoj korisničkoj bazi podataka, tada riskirate sigurnost svojih korisnika – a rizik je veći nego što mislite. Poznata je činjenica da većina korisnika koristi istu lozinku na više stranica, pa čak i ako ne pohranjujete nikakve osjetljive informacije, pristup lozinkama pohranjenim u vašoj bazi podataka može napadaču omogućiti pristup nalozima koje vaši korisnici imaju na drugim stranicama.

Ključ za sigurno pohranjivanje korisničkih lozinki u bazi podataka se oslanja na ne pohranjivanje same lozinke, već na njen hash. Funkcija heširanja lozinke uzima lozinku kao ulaz, dodaje joj nasumične komponente (sol), a zatim na nju primjenjuje nekoliko jednosmernih kriptografskih transformacija. Rezultat je novi niz znakova koji nema sličnosti sa originalnom lozinkom i nema poznati način da se ponovo transformiše u originalnu lozinku. Hešovi lozinki se mogu provjeriti umjesto pravih lozinki jer su funkcije heširanja ponovljive: s obzirom na iste unose (lozinka i sol), rezultat je uvijek isti.



Heširanje lozinke je složen zadatak koji je teško ispraviti. Preporučuje se da ne implementirate vlastito rješenje, već da se oslonite na dobro poznate biblioteke koje je recenzirala zajednica. U sljedećem odjeljku, Werkzeugove funkcije heširanja lozinke će biti demonstrirane. Drugi dobri izbori za heširanje lozinki su [bcrypt](#) i [Passlib](#). Ako ste zainteresirani da saznate šta je uključeno u generiranje sigurnih hashova lozinki, pročitajte članak "[Salted Password Hashing - Doing It Right](#)" od [Defuse Security](#) je vrijedno čitanja.

Haširanje lozinki uz Werkzeug Sigurnosni

modul Werkzeuga praktično implementira sigurno heširanje lozinki. Ova funkcionalnost je izložena sa samo dvije funkcije, koje se koriste u fazama registracije i verifikacije, respektivno:

`generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)`

Ova funkcija uzima lozinku u obliku običnog teksta i vraća hash lozinke kao niz koji se može pohraniti u korisničku bazu podataka. Zadane vrijednosti za method i salt_length su dovoljne za većinu slučajeva upotrebe.

`check_password_hash(hash, lozinka)`

Ova funkcija uzima hash lozinke koji je prethodno pohranjen u bazi podataka i lozinku koju je unio korisnik. Povratna vrijednost True ukazuje da je korisnička lozinka ispravna.

Primer 8-1 pokazuje promene u modelu korisnika kreirane u [poglavlju 5](#) da bi se prilagodilo heširanju lozinke.

Primjer 8-1. app/models.py: heširanje lozinke u modelu korisnika

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model): #
    password_hash =
    db.Column(db.String(128))

    @property
    def password(self):
        podizanje AttributeError('password nije čitljiv atribut')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password): vratiti
        check_password_hash(self.password_hash, password)
```

Funkcija heširanja lozinke implementirana je kroz svojstvo samo za pisanje koje se zove lozinka. Kada je ovo svojstvo postavljeno, metoda postavljanja će pozvati Werkzeugovu funkciju `generate_password_hash()` i upisati rezultat u polje `password_hash`.

Pokušaj čitanja svojstva lozinke će vratiti grešku, jer je jasno da se originalna lozinka ne može oporaviti nakon heširanja.

Metoda `verify_password()` uzima lozinku i prosljeđuje je Werkzeugovoj funkciji `check_password_hash()` radi provjere u odnosu na heširanu verziju pohranjenu u modelu korisnika. Ako ovaj metod vrati True, onda je lozinka ispravna.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 8a da provjerite ovu verziju aplikacije.

Funkcija heširanja lozinke je sada završena i može se testirati u Ijusci:

```
(venv) $ flask shell >>> u
= User() >>> u.password
= 'mačka' >>> u.password
Traceback (posljednji
posljednji poziv):
Fajl "<konzola>", red 1, u <module> Fajl "/"
home/flask/flasky/app/models.py", red 24, u podizanju lozinke
    AttributeError('lozinka nije čitljiv atribut')
AttributeError: lozinka nije čitljiv atribut >>> u.password_hash

'pbkdf2:sha256:50000$moHwFH1B$ef1574909f9c549285e8547cad181c5e0213cf44a4aba40aver ')
```

```
Tačno
>>> u.verify_password('pas')
False
>>> u2 = User() >>>
u2.password = 'mačka' >>>
u2.password hash
```

Obratite pažnju na to kako pokušaj pristupa svojstvu lozinke korisnika vraća AttributeError. Također, korisnici u i u2 imaju potpuno različite hešove lozinki, iako oboje koriste istu lozinku. Kako bi se osiguralo da ova funkcionalnost i dalje radi u budućnosti, prethodni testovi koji su urađeni ručno mogu se napisati kao jedinični testovi koji se mogu lako ponoviti. U [primjeru 8-2](#) prikazan je novi modul unutar paketa testova sa tri nova testa koji koriste nedavne promjene korisničkog modela.

Primjer 8-2. tests/test_user_model.py: testovi heširanja lozinki

```
import unittest iz
app.models import User

class UserModelTestCase(unittest.TestCase): def
    test_password_setter(self): u = User(password =
        'mačka') self.assertTrue(u.password_hash
        nije None)

    def test_no_password_getter(self): u =
        User(password = 'mačka') sa
        self.assertRaisas(AttributeError): u.password

    def test_password_verification(self):
        u = Korisnik(password = 'mačka')
        self.assertTrue(u.verify_password('mačka'))
        self.assertFalse(u.verify_password('dog'))
```

```
def test_password_salts_are_random(self): u =
    Korisnik(password='mačka') u2 =
    Korisnik(password='mačka')
    self.assertTrue(u.password_hash != u2.password_hash)
```

Da pokrenete ove nove testove jedinice, koristite sljedeću naredbu:

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... test_app_is_testing      uredu
(test_basics.BasicsTestCase) ... test_no_password_getter                  uredu
(test_user_model.UserModelTestCase) ... test_password_salts_areModelTestCase) ... uredu
test_password_salts_areMo_random_test. test_password_verification           uredu
(test_user_model.UserModelTestCase) ...                                     uredu
                                         uredu
```

Obavljen 6 testova za 0,379s

uredu

Možete pokrenuti paket za testiranje jedinica na ovaj način svaki put kada želite da potvrđite da sve radi kako se očekuje. Posjedovanje automatizacije čini provjeru ove funkcije vrlo jeftinom, tako da testiranje treba često ponavljati, kako bi se osiguralo da se ova funkcionalnost ne pokvari u budućnosti.

Kreiranje nacrta za autentifikaciju

Nacrti su uvedeni u [Poglavlju 7](#) kao način da se definisi rute u globalnom opsegu nakon što je kreiranje aplikacije prebačeno u fabričku funkciju. U ovom odeljku, rute koje se odnose na podsistem za autentifikaciju korisnika biće dodata drugom nacrtu, koji se zove auth. Korištenje različitih nacrta za različite podsisteme aplikacije odličan je način da kod bude uredno organiziran.

Nacrt autentifikacije će biti smješten u Python paketu s istim imenom. Konstruktor paketa nacrt će kreirati objekt nacrt i uvozi rute iz modula views.py. To je prikazano u [primjeru 8-3](#).

Primjer 8-3. app/auth/__init__.py: kreiranje nacrta za autentifikaciju

```
iz flask import Blueprint

auth = Nacrt('auth', __name__)

od . uvoz pogleda
```

Modul app/auth/views.py, prikazan u [primjeru 8-4](#), uvozi nacrt i definira rute povezane s autentifikacijom koristeći svoj dekorator ruta . Za sada je dodana /login ruta, koja prikazuje predložak čuvara mesta istog imena.

Primjer 8-4. app/auth/views.py: rute nacrt-a autentifikacije i funkcije pregleda

```
iz flask import render_template iz . import
auth

@auth.route('/login') def
login(): return
    render_template('auth/login.html')
```

Imajte na umu da je datoteka šablona data render_template() pohranjena unutar auth direktorija. Ovaj direktorij mora biti kreiran unutar app/templates, jer Flask očekuje da staze šablona budu relativne u odnosu na direktorij predložaka aplikacije. Pohranjivanjem predložaka nacrt-a u vlastiti poddirektorij, ne postoji rizik od sudara imena s glavnim nacrtom ili bilo kojim drugim nacrtima koji će biti dodati u budućnosti.



Nacrti se također mogu konfigurirati da imaju svoje nezavisne direktorije za predloške. Kada je konfiguirano više direktorija šablona, funkcija render_template() prvo pretražuje direktorij predložaka konfiguriran za aplikaciju, a zatim pretražuje direktorije predložaka definirane nacrtima.

Nacrt autentifikacije treba biti priložen aplikaciji u tvorničkoj funkciji create_app(), kao što je prikazano u [primjeru 8-5](#).

Primjer 8-5. app/__init__.py: registracija nacrt-a autentifikacije

```
def create_app(config_name):
    # ...
    iz .auth uvoz auth kao auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    povratna aplikacija
```

Argument url_prefix u registraciji nacrt-a nije obavezan. Kada se koriste, sve rute definisane u nacrtu će biti registrovane sa datim prefiksom, u ovom slučaju /auth. Na primjer, /login ruta će biti registrirana kao /auth/login, a potpuno kvalificirani URL pod razvojnim web serverom tada postaje <http://localhost:5000/auth/login>.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 8b da provjerite ovu verziju aplikacije.

Autentifikacija korisnika sa Flask-Login

Kada se korisnici prijave u aplikaciju, njihovo stanje autentifikacije mora biti zabilježeno u korisničkoj sesiji, tako da se pamti dok se kreću kroz različite stranice.

Flask-Login je mala, ali izuzetno korisna ekstenzija koja je specijalizovana za upravljanje ovim posebnim aspektom sistema za autentifikaciju korisnika, bez vezivanja za određeni mehanizam provjere autentičnosti.

Za početak, ekstenzija treba biti instalirana u virtuelnom okruženju:

```
(venv) $ pip install flask-login
```

Priprema korisničkog modela za prijave Flask-Login

blisko radi sa korisničkim objektima aplikacije. Da bi mogao da radi sa korisničkim modelom aplikacije, ekstenzija Flask-Login zahteva da implementira nekoliko uobičajenih svojstava i metoda. Potrebne stavke prikazane su u [tabeli 8-1](#).

Tabela 8-1. Flask-Login potrebne stavke

Svojstvo/metoda	Opis
is_authenticated	Mora biti istinito ako korisnik ima važeće vjerodajnice za prijavu ili False u suprotnom.
je_aktivan	Mora biti True ako je korisniku dozvoljeno da se prijavi ili False u suprotnom. Vrijednost False se može koristiti za onemogućene račune.
je_anonimno	Uvijek mora biti False za obične korisnike i True za poseban korisnički objekt koji predstavlja anonimne korisnike.
get_id()	Mora vratiti jedinstveni identifikator za korisnika, kodiran kao Unicode niz.

Ova svojstva i metode mogu se implementirati direktno u klasi modela, ali kao lakšu alternativu Flask-Login pruža klasu UserMixin koja ima zadane implementacije koje su prikladne za većinu slučajeva. Ažurirani model korisnika prikazan je u [primjeru 8-6](#).

Primjer 8-6. app/models.py: ažuriranja modela korisnika za podršku prijavljivanja korisnika

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(64), unique=True, index=True)
    korisničko_ime = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Imajte na umu da je dodano i polje e-pošte . U ovoj aplikaciji, korisnici će se prijaviti sa svojim email adresama, jer je manje vjerovatno da će ih zaboraviti nego njihova korisnička imena.

Flask-Login se inicijalizira u tvorničkoj funkciji aplikacije, kao što je prikazano u [primjeru 8-7](#).

Primjer 8-7. app/_init_.py: Inicijalizacija Flask-Login

```
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app) #
    ...
```

Atribut login_view objekta LoginManager postavlja krajnju tačku za stranicu za prijavu. Flask-Login će preusmjeriti na stranicu za prijavu kada anonimni korisnik pokuša pristupiti zaštićenoj stranici. Budući da je ruta za prijavu unutar nacrtta, ona mora imati prefiks naziva nacrtta.

Konačno, Flask-Login zahtijeva od aplikacije da odredi funkciju koja će biti pozvana kada ekstenzija treba da učita korisnika iz baze podataka s obzirom na njegov identifikator. Ova funkcija je prikazana u [primjeru 8-8](#).

Primjer 8-8. app/models.py: funkcija učitavanja korisnika

```
od . import login_manager

@login_manager.user_loader
def load_user(user_id):
    vrati User.query.get(int(user_id))
```

Dekorator login_manager.user_loader se koristi za registraciju funkcije sa Flask-Login, koji će je pozvati kada treba da preuzme informacije o prijavljenom korisniku. Identifikator korisnika će biti proslijeđen kao string, tako da ga funkcija pretvara u cijeli broj prije nego što ga proslijedi upitu Flask-SQLAlchemy koji učitava korisnika. Povratna vrijednost funkcije mora biti korisnički objekt ili None ako je identifikator korisnika nevažeći ili se dogodila bilo koja druga greška.

Zaštita ruta Za zaštitu

rute tako da joj mogu pristupiti samo provjereni korisnici, Flask-Login pruža dekorator login_required . Slijedi primjer njegove upotrebe:

```
from flask_login import login_required

@app.route('/secret')
@login_required def
secret(): return 'Samo
provjereni korisnici su dozvoljeni!'
```

Iz ovog primjera možete vidjeti da je moguće „ulančati“ višestruke funkcije dekoratora. Kada se dva ili više dekoratora dodaju funkciji, svaki dekorator utječe samo na one koji su ispod njega, pored ciljne funkcije. U ovom primjeru, funkcija `secret()` će biti zaštićena od neovlaštenih korisnika sa `login_required`, a zatim će rezultirajuća funkcija biti registrirana u Flask kao ruta. Obrnuti redoslijed će proizvesti pogrešan rezultat, jer će originalna funkcija biti registrirana kao ruta prije nego što primi dodatna svojstva od dekoratora `login_required`.

Zahvaljujući dekoratoru `login_required`, ako ovoj ruti pristupi korisnik koji nije autentificiran, Flask-Login će presresti zahtjev i umjesto toga poslati korisnika na stranicu za prijavu.

Dodavanje obrasca za prijavu

Obrazac za prijavu koji će biti predstavljen korisnicima ima tekstualno polje za adresu e-pošte, polje za lozinku, polje za potvrdu „zapamti me“ i dugme za slanje. Klasa obrasca Flask-WTF koja definira ovaj oblik prikazana je u [primjeru 8-9](#).

Primjer 8-9. app/auth/forms.py: obrazac za prijavu

```
from flask_wtf import FlaskForm iz
wtforms import StringField, PasswordField, BooleanField, SubmitField iz wtforms.validators
import DataRequired, Length, Email

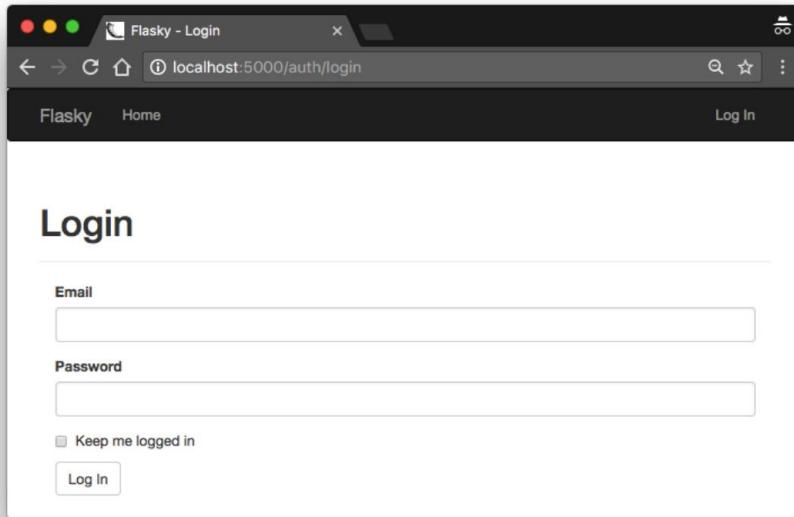
class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    Remember_me
    = BooleanField ('Zadrži me prijavljenim') submit = SubmitField('Prijava')
```

Klasa `PasswordField` predstavlja element `<input>` sa `type="password"`. Klasa `BooleanField` predstavlja potvrđni okvir.

Polje e-pošte koristi validatore `Length()` i `Email()` iz WTForms pored `DataRequired()`, kako bi se osiguralo da korisnik ne samo da daje vrijednost za ovo polje, već da je ona važeća. Prilikom pružanja liste validatora, WTForms će ih procijeniti navedenim redoslijedom, a u slučaju neuspjeha validacije prikazana poruka o grešci će biti jedna od prvih validatora koji nisu uspjeli.

Predložak povezan sa stranicom za prijavu pohranjen je u auth/login.html. Ovaj predložak samo treba da prikaže obrazac koristeći Flask-Bootstrapov wtf.quick_form() makro.

Slika 8-1 prikazuje obrazac za prijavu koji prikazuje web pretraživač.



Slika 8-1. Obrazac za prijavu

Traka za navigaciju u predlošku base.html koristi Jinja2 uslov za prikaz linkova „Prijava“ ili „Odjava“ u zavisnosti od statusa prijavljenog trenutnog korisnika. Uslov je prikazan u primjeru 8-10.

Primjer 8-10. app/templates/base.html: Veze navigacijske trake za prijavu i odjavu

```
<ul class="nav navbar-nav navbar-right">
  {% if current_user.is_authenticated %} <li><a
    href="{{ url_for('auth.logout') }}>Odjava </a></li> {% else %} <li><a
    href="{{ url_for('auth.login') }}>Prijava </a></li> {% endif %} </ul>
```

Varijabla current_user koja se koristi u uvjetu definirana je od strane Flask-Login-a i automatski je dostupna za pregled funkcija i predložaka. Ova varijabla sadrži trenutno prijavljenog korisnika ili proxy anonimni korisnički objekat ako korisnik nije prijavljen. Anonimni korisnički objekti imaju svojstvo is_authenticated postavljeno na False, tako da

izraz current_user.is_authenticated je zgodan način da saznote da li je trenutni korisnik prijavljen.

Potpisivanje korisnika

u Implementacija funkcije prikaza login() prikazana je u **primjeru 8-11**.

Primjer 8-11. app/auth/views.py: ruta za prijavu

```
from flask import render_template, redirect, request, url_for, flash from flask_login
import login_user from . import auth from ..models import User from .forms import
LoginForm

@auth.route('/login', methods=['GET', 'POST']) def login():
form = LoginForm() if form.validate_on_submit():

    user = User.query.filter_by(email=form.email.data).first() ako korisnik
    nije Ništa i user.verify_password(form.password.data): login_user(user,
        form.remember_me.data) next = request.args.get('next') ako je sljedeće
    Ništa ili nije next.startswith('/'): next = url_for('main.index') return
    redirect(next) flash('Nevažeće korisničko ime ili lozinka.') return .
    render_template('auth/login.html', form=form)
```

Funkcija view kreira objekat LoginForm i koristi ga kao jednostavan obrazac u **poglavlju 4**. Kada je zahtjev tipa GET, funkcija prikaza samo prikazuje predložak, koji zauzvrat prikazuje obrazac. Kada se obrazac pošalje u POST zahtjevu, funkcija validate_on_submit () Flask-WTF-a provjerava valjanost varijabli obrasca, a zatim pokušava prijaviti korisnika.

Za prijavu korisnika, funkcija počinje učitavanjem korisnika iz baze podataka pomoću e-pošte koja se nalazi uz obrazac. Ako korisnik sa datom adresom e-pošte postoji, tada se poziva njegova metoda verify_password() sa lozinkom koja je također došla uz obrazac. Ako je lozinka ispravna, funkcija Flask-Login login_user() se poziva da zabilježi korisnika kao prijavljenog za korisničku sesiju. Funkcija login_user() vodi korisnika da se prijavi i opcioni "zapamti me" Boolean, koji je također dostavljen uz obrazac. Vrijednost False za ovaj argument uzrokuje da korisnička sesija istekne kada se prozor pretraživača zatvori, tako da će se korisnik morati ponovo prijaviti sljedeći put. Vrijednost True uzrokuje da se dugoročni kolačić postavi u pretraživač korisnika, koji Flask-Login koristi za vraćanje korisničke sesije. Opciona opcija konfiguracije REMEMBER_COOKIE_DURATION može se koristiti za promjenu zadanoj jednogodišnjeg trajanja kolačića za pamćenje.

U skladu sa obrascem Post/Redirect/Get o kojem se govorи u pogлављу 4, POST zahtjev koji je posao vjerodajnice za prijavu završava se preusmjeravanjem, ali postoje dvije moguće URL destinacije. Ako je obrazac za prijavu predstavljen korisniku kako bi se spriječio neovlašteni pristup zaštićenom URL-u koji je korisnik želio posjetiti, tada će Flask-Login sačuvati taj originalni URL u sljedećem argumentu niza upita, kojem se može pristupiti iz request.args rječnika . Ako sljedeći argument stringa upita nije dostupan, umjesto toga se izdaje preusmjeravanje na početnu stranicu. URL u sljedećem se provjerava kako bi se uvjerojao da je relativan URL, kako bi se spriječio zlonamjerni korisnik da koristi ovaj argument za preusmjeravanje nesuđenih korisnika na drugu stranicu.

U slučaju kada je adresa e-pošte ili lozinka koju je dao korisnik nevažeća, postavlja se flash poruka i obrazac se ponovo prikazuje kako bi korisnik ponovo pokušao.



Na proizvodnom serveru, aplikacija mora biti dostupna preko sigurnog HTTP-a, tako da se vjerodajnice za prijavu i korisničke sesije uvijek prenose šifrovane. Bez sigurnog HTTP-a, napadač može presresti osjetljive podatke tokom tranzita.

Predložak za prijavu treba ažurirati da bi se obrazac prikazao. Ove promjene su prikazane u [primjeru 8-12](#).

Primjer 8-12. app/templates/auth/login.html: predložak obrasca za prijavu

```
{% proširuje "base.html" %} {%
import "bootstrap/wtf.html" kao wtf %}

{% block title %}Flasky - Prijava{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Prijava</h1> </div>
<div class="col-md-4">
{{ wtf.quick_form(form) }} </
    div> {% endblock %}
```

Odjava korisnika

Implementacija putanje odjave prikazana je u [primjeru 8-13](#).

Primjer 8-13. app/auth/views.py: ruta za odjavu

```
from flask_login import logout_user, login_required

@auth.route('/logout')
```

```
@login_required
def logout():
    logout_user()
    flash('Odjavljeni ste.') return
    redirect(url_for('main.index'))
```

Za odjavu korisnika, Flask-Login-ova logout_user() funkcija se poziva da ukloni i resetuje korisničku sesiju. Odjava se završava flash porukom koja potvrđuje akciju i preusmjeravanjem na početnu stranicu.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 8c da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod. Da biste bili sigurni da imate instalirane sve ovisnosti, također pokrenite pip install -r requirements.txt.

Razumevanje kako Flask-Login funkcioniše . Flask-

Login je prilično mala ekstenzija, ali zbog mnogih pokretnih delova uključenih u tok autentifikacije, korisnici Flaska često imaju problema da razumeju kako ekstenzija funkcioniše. Sledi redosled operacija koje se dešavaju kada se korisnik prijavi na sistem:

1. Korisnik se kreće na <http://localhost:5000/auth/login> klikom na "Prijava" veza. Rukovalac za ovaj URL vraća predložak obrasca za prijavu.
2. Korisnik unosi svoje korisničko ime i lozinku, te pritisne dugme Pošalji. Ponovo se poziva isti rukovalac, ali sada kao POST zahtjev umjesto GET. a. Rukovalac provjerava valjanost vjerodajnica dostavljenih uz obrazac, a zatim poziva Flask-Login- ovu funkciju login_user() za prijavu korisnika.
 - b. Funkcija login_user() zapisuje ID korisnika u korisničku sesiju kao a string.
 - c. Funkcija pregleda se vraća s preusmjeravanjem na početnu stranicu.
3. Pretraživač prima preusmjeravanje i traži početnu stranicu. a. Poziva se funkcija pregleda za početnu stranicu i ona pokreće renderiranje glavnog Jinja2 šablona.
 - b. Tokom prikazivanja šablona Jinja2, referenca na Flask-Login trenutni_user se pojavljuje po prvi put.
 - c. Kontekst varijabli current_user još nema dodijeljenu vrijednost za ovaj zahtjev, tako da poziva internu funkciju Flask-Login-a _get_user() da sazna ko je korisnik.

- d. Funkcija `_get_user()` provjerava da li postoji korisnički ID pohranjen u korisničkoj sesiji. Ako ne postoji, vraća instancu Flask-Login-ovog `AnonymousUser`.
- Ako postoji ID, on poziva funkciju koju je aplikacija registrirala u dekoratoru `user_loader`, s ID-om kao argumentom.
- e. Rukovalac `user_loader` aplikacije čita korisnika iz baze podataka i vraća ga. Flask-Login ga dodeljuje promenljivoj konteksta `current_user` za trenutni zahtev.
- f. Šablon prima novododijeljenu vrijednost `current_user`.

Dekorator `login_required` se nadograđuje na kontekstu varijable `current_user` tako što dozvoljava da se funkcija uređenog pogleda pokrene samo kada je izraz `current_user.is_authenticated` Tačan . Funkcija `logout_user()` jednostavno briše korisnički ID iz korisničke sesije.

Testiranje prijava

Da bi se potvrdilo da funkcionalnost prijave radi, početna stranica se može ažurirati da pozdravi prijavljenog korisnika imenom. Odeljak šablona koji generiše pozdrav je prikazan u [Primeru 8-14](#).

Primjer 8-14. app/templates/index.html: pozdravljanje prijavljenog korisnika

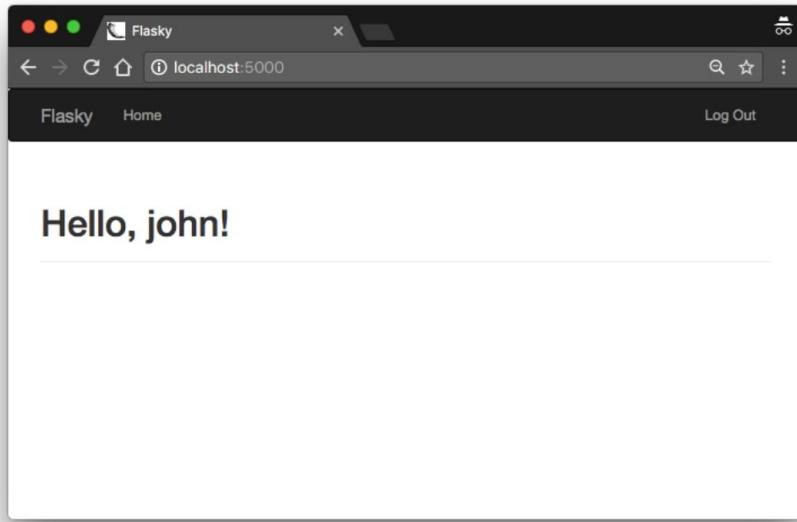
```
Zdravo,
{% if current_user.is_authenticated %}
    {{ current_user.username }} %
else %
    Stranac
{% endif %}!
```

U ovom predlošku se još jednom koristi `current_user.is_authenticated` za određivanje da li je korisnik prijavljen.

Budući da nije izgrađena funkcija registracije korisnika, novi korisnik se u ovom trenutku može registrovati samo iz ljudske:

```
(venv) $ $ flask shell >>> u
= Korisnik(email='john@example.com', korisničko_ime='john', lozinka='mačka') >>>
db.session.add(u) >>> db.session.commit()
```

Prethodno kreirani korisnik sada se može prijaviti. [Slika 8-2](#) prikazuje početnu stranicu aplikacije s prijavljenim korisnikom.



Slika 8-2. Početna stranica je uspješna prijava

Registracija novog korisnika

Kada novi korisnici žele da postanu članovi aplikacije, moraju se registrovati na nju kako bi bili poznati i mogli se prijaviti. Link na stranici za prijavu će ih poslati na stranicu za registraciju, gdje mogu unijeti svoju email adresu, korisničko ime, i lozinku.

Dodavanje obrasca za registraciju korisnika

Obrazac koji će se koristiti na stranici za registraciju traži od korisnika da unese adresu e-pošte, korisničko ime i lozinku. Ovaj obrazac je prikazan u **primjeru 8-15.**

Primjer 8-15. app/auth/forms.py: obrazac za registraciju korisnika

```
from flask_wtf import FlaskForm iz
wtforms import StringField, PasswordField, BooleanField, SubmitField iz wtforms.validators
import DataRequired, Length, Email, Regexp, EqualTo iz wtforms import ValidationError
iz ..models import User

class RegistrationForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
```

```

korisničko ime = StringField('Username',
    validators=[ DataRequired(), Length(1, 64), Regexp('^[A-
    Za-z][A-Za-z0-9_]*$', 0,
        'Korisnička imena moraju imati samo slova, brojeve, tačke ili
        'podvlake')]) lozinka = PasswordField('Password', validators=[

        DataRequired(), EqualTo('password2', message='Lozinke se moraju podudarati.')])
password2 = PasswordField('Potvrdi lozinku', validators=[DataRequired()]) submit =
SubmitField('Registriraj se')

def validate_email(self, field):
    User.query.filter_by(email=field.data).first():
        raise ValidationError('E-pošta je već registravana.')

def validate_username(self, field):
    User.query.filter_by(username=field.data).first(): raise
        ValidationError('Korisničko ime je već u upotrebi.')

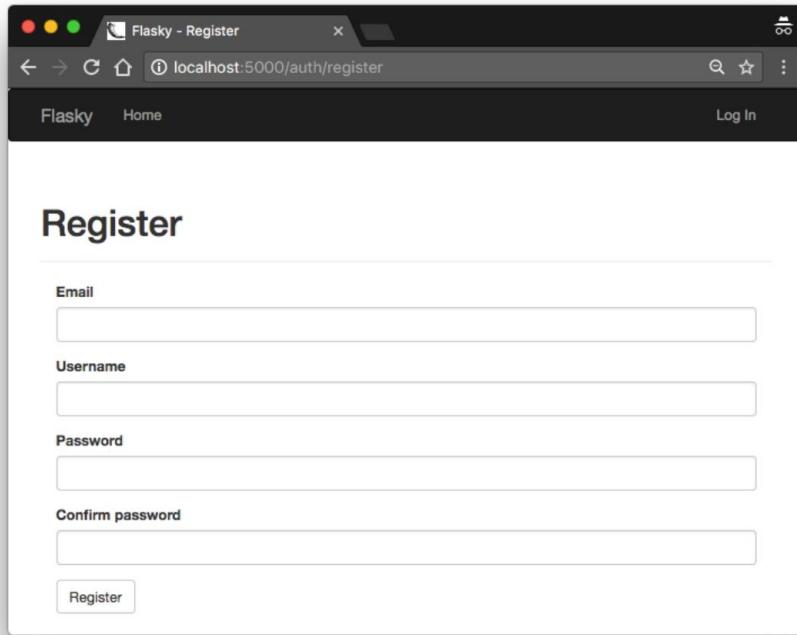
```

Ovaj obrazac koristi validator Regexp iz WTForms kako bi osigurao da polje korisničkog imena počinje slovom i da sadrži samo slova, brojeve, donje crte i tačke. Dva argumenta za validator koji prate regularni izraz su zastavice regularnog izraza i poruka o grešci koja se prikazuje u slučaju neuspjeha.

Lozinka se unosi dva puta kao sigurnosna mjera, ali ovaj korak čini neophodnim da se provjeri da li dva polja lozinke imaju isti sadržaj, što se radi s drugim validatorom iz WTForms koji se zove EqualTo. Ovaj validator je vezan za jedno od polja lozinke sa imenom drugog polja datim kao argument.

Ovaj obrazac također ima dva prilagođena validatora implementirana kao metode. Kada obrazac definiše metodu sa prefiksom validate_ iza kojeg sledi ime polja, metoda se poziva kao dodatak svim redovno definisanim validatorima. U ovom slučaju, prilagođeni validatori za e-poštu i korisničko ime osiguravaju da date vrijednosti nisu duplikati. Prilagođeni validatori ukazuju na grešku validacije podizanjem izuzetka ValidationError sa tekstom poruke o grešci kao argumentom.

Predložak koji predstavlja ovaj obrazac naziva se /templates/auth/register.html. Kao i predložak za prijavu, ovaj također prikazuje obrazac sa wtf.quick_form(). Stranica za registraciju je prikazana na [slici 8-3](#).



Slika 8-3. Novi obrazac za registraciju korisnika

Stranica za registraciju mora biti povezana sa stranicu za prijavu kako bi je korisnici koji nemaju račun mogli lako pronaći. Ova promjena je prikazana u **primjeru 8-16**.

Primjer 8-16. app/templates/auth/login.html: link na stranicu za registraciju

```
<p>
    Novi korisnik?
    <a href="{{ url_for('auth.register') }}"> Kliknite ovdje za
    registraciju </a> </p>
```

Registracija novih korisnika

Rukovanje registracijama korisnika ne predstavlja velika iznenađenja. Kada je obrazac za registraciju dostavljen i potvrđen, novi korisnik se dodaje u bazu podataka koristeći podatke koje je korisnik pružio. Funkcija pogleda koja izvodi ovaj zadatak prikazana je u **primjeru 8-17**.

Primjer 8-17. app/auth/views.py: ruta registracije korisnika

```
@auth.route('/register', methods=['GET', 'POST']) def register():
    form = RegistrationForm() if form.validate_on_submit():
        korisnik = Korisnik(email=form.email.data , korisničko
                            ime=form.username.data, lozinka=form.password.data)
        db.session.add(user) db.session.commit()
        flash('Sada se možete prijaviti.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html',
                           form=form)
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 8d da provjerite ovu verziju aplikacije.

Account Confirmation

Za određene vrste aplikacija, važno je osigurati da su podaci o korisniku koji su dati prilikom registracije valjani. Uobičajeni zahtjev je osigurati da se korisnik može kontaktirati putem navedene adrese e-pošte.

Da bi potvrdili adresu e-pošte, aplikacije šalju potvrdni email korisnicima odmah nakon registracije. Novi nalog je inicijalno označen kao nepotvrđen sve dok se ne prate uputstva u mejlu, što dokazuje da je korisnik primio e-poštu. Procedura potvrde naloga obično uključuje klikanje na posebno kreiranu URL vezu koja uključuje token za potvrdu.

Generiranje Confirmation tokena sa svojim opasnim Najjednostavniji

link za potvrdu naloga bi bio URL u formatu `http://www.example.com/auth/confrm/<id>` uključen u e-poruku za potvrdu, gdje je `<id>` numerički ID koji je dodijeljen korisnika u bazi podataka. Kada korisnik klikne na vezu, funkcija pregleda koja upravlja ovom rutom prima korisnički ID za potvrdu kao argument i može lako ažurirati potvrđeni status korisnika.

Ali ovo očigledno nije sigurna implementacija, jer će svaki korisnik koji shvati format linkova za potvrdu moći potvrditi proizvoljne račune samo slanjem nasumičnih brojeva u URL-u. Ideja je da se `<id>` u URL-u zamjeni tokenom koji sadrži iste informacije, ali na takav način da samo server može generirati važeće URL-ove za potvrdu.

Ako se prisjetite rasprave o korisničkim sesijama u poglavju 4, Flask koristi kriptografski potpisane kolačice kako bi zaštitio sadržaj korisničkih sesija od neovlaštenog pristupa. Kolačići korisničke sesije sadrže kriptografski potpis koji generiše paket koji se naziva itsdangerous. Ako se sadržaj korisničke sesije promjeni, potpis se više neće podudarati sa sadržajem, pa Flask odbacuje sesiju i započinje novu. Isti koncept se može primijeniti na tokene za potvrdu.

Slijedi kratka sesija ljske koja pokazuje kako itsdangerous može generirati potpisani token koji sadrži korisnički ID unutar:

```
(venv) $ flask shell >>> iz
svog opasnog uvoza TimedJSONWebSignatureSerializer kao serijalizator >>> s =
Serializer(app.config['SECRET_KEY'], expires_in=3600) >>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.eyJ ...' >>>
podaci = s.loads(token) >>> podaci {'confirm': 23}
```

Njegov dangerous paket nudi nekoliko tipova generatora tokena. Među njima, klasa TimedJSONWebSignatureSerializer generiše JSON Web potpise (JWS) sa vremenskim istekom. Konstruktor ove klase uzima ključ za šifriranje kao argument, koji u Flask aplikaciji može biti konfigurirani SECRET_KEY.

Dumps () metoda generiše kriptografski potpis za podatke date kao argument, a zatim serijalizira podatke plus potpis kao pogodan niz tokena.

Argument expires_in postavlja vrijeme isteka za token, izraženo u sekundama.

Za dekodiranje tokena, objekat serijalizatora pruža metodu loads() koja uzima token kao jedini argument. Funkcija provjerava potpis i vrijeme isteka i, ako su oboje valjani, vraća originalne podatke. Kada se metodi loads() dodijeli nevažeći token ili važeći token koji je istekao, stvara se izuzetak.

Generisanje tokena i verifikacija pomoću ove funkcionalnosti mogu se dodati korisničkom modelu. Promjene su prikazane u **primjeru 8-18**.

Primjer 8-18. app/models.py: potvrda korisničkog naloga

```
iz svog opasnog uvoza TimedJSONWebSignatureSerializer kao serijalizator iz flask import
current_app iz . import db
```

```
class User(UserMixin, db.Model):
    ...
    # potvrđeno = db.Column(db.Boolean, default=False)
```

```

def generate_confirmation_token(self, expiration=3600):
    s = Serializer(current_app.config['SECRET_KEY'], expiration )
    return s.dumps({'confirm': self.id}).decode('utf-8')

def potvrди (self, token):
    s = Serializer(current_app.config['SECRET_KEY']) pokušaj:
    podaci = s.loads(token.encode('utf-8')) osim: vrati False

    if data.get('confirm') != self.id:
        return False
    self.confirmed = Tačno
    db.session.add(self)
    return Tačno

```

Generiranje_confirmation_token () metoda generiše token sa zadanim vremenom valjanosti od jednog sata. Metoda confirm() provjerava token i, ako je važeća, postavlja novi potvrđeni atribut u korisničkom modelu na Tačno.

Osim provjere tokena, funkcija confirm() provjerava da li id iz tokena odgovara prijavljenom korisniku, koji je pohranjen u current_user. Ovo osigurava da se token za potvrdu za određenog korisnika ne može koristiti za potvrdu drugog korisnika.



Budući da je modelu dodana nova kolona za praćenje potvrđenog stanja svakog naloga, potrebno je generirati i primijeniti novu migraciju baze podataka.

Dvije nove metode dodane modelu korisnika lako se testiraju u jediničnim testovima. Jedinične testove možete pronaći u GitHub spremištu za aplikaciju.

Slanje potvrđnih e-poruka Trenutna /

register ruta se preusmjerava na /index nakon dodavanja novog korisnika u bazu podataka. Prije preusmjeravanja, ova ruta sada mora poslati e-poruku s potvrdom. Ova promjena je prikazana u **primjeru 8-19.**

Primjer 8-19. app/auth/views.py: ruta registracije s potvrđnom e-poštrom

```

iz ..email import send_email

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():

        # ...

```

```

db.session.add(user)
db.session.commit()
token = user.generate_confirmation_token()
send_email(user.email, 'Potvrdite svoj račun', 'auth/email/
    confirm', user=user, token=token) flash(' Potvrda e-pošte
vam je poslana e-poštom.') return redirect(url_for('main.index')) return
render_template('auth/register.html', form=form)

```

Imajte na umu da je poziv db.session.commit() morao biti dodan prije slanja e-pošte s potvrdom. Problem je u tome što se novim korisnicima dodeljuje id kada se predaju bazi podataka, a ovaj id je potreban za generisanje tokena za potvrdu.

Predlošci e-pošte koje koristi nacrt autentikacije bit će dodati u direktorij templates/auth/email kako bi bili odvojeni od HTML šablona. Kao što je diskutovano u [Poglavlju 6](#), za svaku e-poruku su potrebna dva šablonu za običan tekst i HTML verziju tijela. Kao primjer, [primjer 8-20](#) prikazuje verziju otvorenog teksta šablonu e-pošte za potvrdu, a ekvivalentnu HTML verziju možete pronaći u GitHub spremištu.

Primjer 8-20. app/templates/auth/email/conrm.txt: tekstualno tijelo potvrđne e-pošte

Poštovani {{ user.username }},

Dobrodošli u Flasky!

Da potvrdite svoj račun kliknite na sljedeći link:

`{{ url_for('auth.confirm', token=token, _external=True) }}`

S poštovanjem,

Flasky Team

Napomena: odgovori na ovu adresu e-pošte se ne prate.

Po defaultu, url_for() generira relativne URL-ove; tako, na primjer, url_for('auth.confirm', token='abc') vraća string '/auth/confirm/abc'.

Ovo, naravno, nije važeći URL koji se može poslati u e-poruci, jer je to samo dio putanje URL-a. Relativni URL-ovi rade dobro kada se koriste u kontekstu web stranice jer ih pretraživač konvertuje u apsolutne URL-ove dodavanjem imena hosta i broja porta sa trenutne stranice, ali kada se URL šalje putem e-pošte ne postoji takav kontekst.

Argument _external=True se dodaje pozivu url_for() kako bi se zatražio potpuno kvalificirani URL koji uključuje shemu (<http://> ili <https://>), ime hosta i port.

Funkcija pregleda koja potvrđuje račune prikazana je u [primjeru 8-21](#).

Primjer 8-21. app/auth/views.py: potvrda korisničkog naloga

```
from flask_login import current_user

@auth.route('/confirm/<token>')
@login_required def potvrdi(token):
    ako trenutni_user.confirmed: vratи
        redirect(url_for('main.index')) ako
            trenutni_user.confirm(token): db.session .commit()
        flash(' Potvrdili ste svoj nalog. Hvala!') else: flash('
            Link za potvrdu je nevažeći ili je istekao.') return
        redirect(url_for('main.index'))
```

Ova ruta je zaštićena dekoratorom login_required iz Flask-Login-a, tako da kada korisnici kliknu na link iz e-poruke za potvrdu od njih se traži da se prijave prije nego dođu do ove funkcije pregleda.

Funkcija prvo provjerava da li je prijavljen korisnik već potvrđen i u tom slučaju se preusmjerava na početnu stranicu, jer očito nema šta da se radi. Ovo može sprječiti nepotreban rad ako korisnik greškom klikne na token više puta.

Budući da se stvarna potvrda tokena vrši u potpunosti u modelu korisnika , sve što funkcija pregleda treba da uradi je da pozove metodu confirm() i zatim da treperi poruku u skladu sa rezultatom. Kada potvrda uspije, potvrđeni atribut korisničkog modela se mijenja i dodaje sesiji, a zatim se urezuje sesija baze podataka.

Svaka aplikacija može odlučiti šta je dozvoljeno nepotvrđenim korisnicima prije nego što potvrde svoje račune. Jedna od mogućnosti je da se dozvoli nepotvrđenim korisnicima da se prijave, ali im se pokaže samo stranica na kojoj se od njih traži da potvrde svoje račune prije nego što dobiju daljnji pristup.

Ovaj korak se može uraditi koristeći Flask-ov pre_request zakačicu , koja je ukratko opisana u poglavljju 2. Iz nacrta, zakačica before_request se primjenjuje samo na zahtjeve koji pripadaju nacrtu. Da biste instalirali zakačicu za nacrt za sve zahtjeve aplikacije, umjesto toga se mora koristiti dekorater before_app_request . **Primjer 8-22** pokazuje kako je implementiran ovaj rukovalac.

Primjer 8-22. app/auth/views.py: filtriranje nepotvrđenih računa pomoću obradivač before_app_request

```
@auth.before_app_request
def before_request(): ako je
    current_user.is_authenticated \ a ne
        current_user.confirmed \ i
        request.blueprint != 'auth' \ i
        request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed') def
unconfirmed(): ako je
    trenutni_user.is_anonymous ili current_user.confirmed : return
        redirect(url_for('main.index')) return render_template('auth/
unconfirmed.html')
```

Obradivač before_app_request će presresti zahtjev kada postoje tri uslova tačno:

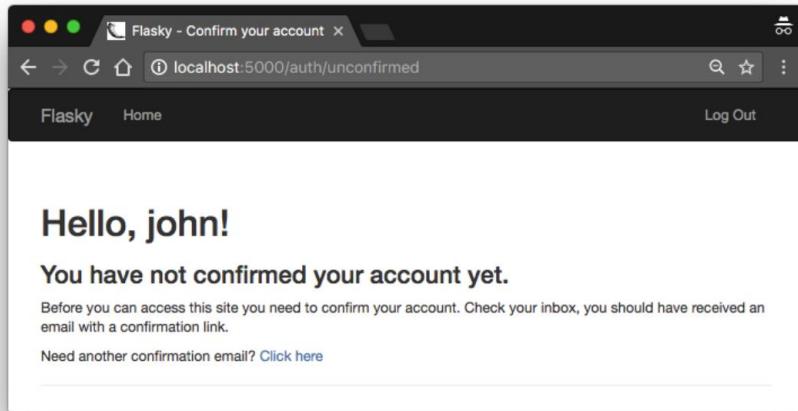
1. Korisnik je prijavljen (current_user.is_authenticated je True).
2. Nalog za korisnika nije potvrđen.
3. Traženi URL je izvan plana provjere autentičnosti i nije za statičku datoteku. Pristup autentifikacijskim rutama je potrebno odobriti, jer su to rute koje će omogućiti korisniku da potvrdi nalog ili izvrši druge funkcije upravljanja naloga.

Ako su ova tri uslova ispunjena, tada se izdaje preusmjeravanje na novu /auth/unconrmed rutu koja prikazuje stranicu sa informacijama o potvrdi naloga.



Kada povratni poziv before_request ili before_app_request vrati odgovor ili preusmjeravanje, Flask to šalje klijentu bez pozivanja funkcije pregleda koja je povezana sa zahtjevom. Ovo efektivno omogućava ovim povratnim pozivima da presretnu zahtjev kada je to potrebno.

Stranica koja se prikazuje nepotvrđenim korisnicima (pričuvano na [slici 8-4](#)) samo prikazuje predložak koji korisnicima daje upute kako da potvrde svoje račune i nudi vezu za traženje nove e-pošte za potvrdu, u slučaju da je originalna e-pošta izgubljena. Ruta koja ponovo šalje e-poruku za potvrdu prikazana je u [primjeru 8-23](#).



Slika 8-4. Nepotvrđena stranica naloga

Primjer 8-23. app/auth/views.py: ponovno slanje e-pošte za potvrdu naloga

```
@auth.route('/confirm')
@login_required def
resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email(current_user.email, 'Potvrdite svoj račun',
               'auth/email/confirm', user=current_user, token=token) flash(
        'Poslana vam je nova e-pošta za potvrdu putem e-pošte.') return
    redirect(url_for('main.index'))
```

Ova ruta ponavlja ono što je urađeno na ruti registracije koristeći `current_user`, korisnika koji je prijavljen, kao ciljnog korisnika. Ova ruta je također zaštićena `login_required` kako bi se osiguralo da kada joj se pristupi, korisnik koji postavlja zahtjev je autentificiran.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 8e da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

Upravljanje računa

Korisnici koji imaju račune s aplikacijom možda će morati s vremena na vrijeme da unesu izmjene na svoje račune. Sljedeći zadaci mogu se dodati nacrtu provjere autentičnosti koristeći tehnike predstavljene u ovom poglavlju:

Ažuriranje lozinki

Korisnici koji su svjesni sigurnosti mogu povremeno htjeti promijeniti svoje lozinke.

Ovo je laka za implementaciju, jer sve dok je korisnik prijavljen, sigurno je predstaviti obrazac koji traži staru lozinku i novu lozinku za zamjenu.

Ova funkcija je implementirana kao urezivanje 8f u GitHub spremištu. Kao dio ove promjene, veza „Odjava“ na traci za navigaciju pretvorena je u padajući meni koji sadrži veze „Promjeni lozinku“ i „Odjavi se“.

Lozinka resetuje

Da biste izbjegli zaključavanje korisnika iz aplikacije kada zaborave svoje lozinke, može se ponuditi opcija resetiranja lozinke. Za implementaciju resetiranja lozinke na siguran način, potrebno je koristiti tokene slične onima koji se koriste za potvrđivanje naloga. Kada korisnik zatraži resetiranje lozinke, na registriranu adresu e-pošte šalje se e-mail sa tokenom za resetiranje. Korisnik zatim klikne na link u e-poruci i, nakon što je token verificiran, prikazuje se obrazac u koji se može unijeti nova lozinka. Ova funkcija je implementirana kao urezivanje 8g u GitHub spremištu.

Promjene adrese e-

pošte Korisnicima se može dati mogućnost da promijene svoju registrovanu adresu e-pošte, ali prije nego što se nova adresa prihvati, ona mora biti potvrđena putem e-pošte za potvrdu. Da bi koristio ovu funkciju, korisnik unosi novu adresu e-pošte u obrazac. Za potvrdu adrese e-pošte, token se šalje e-poštom na tu adresu. Kada server primi token natrag, može ažurirati korisnički objekt. Dok server čeka da primi token, on može pohraniti novu adresu e-pošte u novo polje baze podataka rezervirano za adrese e-pošte na čekanju ili može pohraniti adresu u token zajedno sa ID-om. Ova funkcija je implementirana kao urezivanje 8h u GitHub spremištu.

U sljedećem poglavlju, korisnički podsistem Flaskyja će biti proširen korištenjem korisničkih uloga.

POGLAVLJE 9

Uloge korisnika

Nisu svi korisnici web aplikacija stvorenji jednaki. U većini aplikacija, malom procentu korisnika se povjeravaju dodatne ovlasti koje pomažu da aplikacija radi nesmetano. Administratori su najbolji primjer, ali u mnogim slučajevima postoje i napredni korisnici srednjeg nivoa, poput moderatora sadržaja. Da bi se ovo implementiralo, svim korisnicima je dodijeljena uloga.

Postoji nekoliko načina za implementaciju uloga u aplikaciji. Odgovarajuća metoda uvelike ovisi o tome koliko uloga treba podržati i koliko su razrađene.

Na primjer, jednostavnoj aplikaciji mogu biti potrebne samo dvije uloge, jedna za obične korisnike i jedna za administratore. U ovom slučaju, postojanje logičkog polja `is_administrator` u modelu korisnika može biti sve što je potrebno. Složenijoj aplikaciji mogu biti potrebne dodatne uloge sa različitim nivoima moći između običnih korisnika i administratora. U nekim aplikacijama možda nema smisla ni govoriti o diskretnim ulogama, a umjesto toga davanje skupa pojedinačnih dozvola korisnicima može biti pravi pristup.

Implementacija korisničke uloge predstavljena u ovom poglavlju je hibrid između diskretnih uloga i dozvola. Korisnicima je dodijeljena diskretna uloga, ali svaka uloga definira koje radnje dozvoljava svojim korisnicima da izvode putem liste dozvola.

Reprezentacija uloga u bazi podataka

Jednostavna tabela uloga kreirana je u [poglavlju 5](#) kao sredstvo za demonstriranje odnosa jedan prema više. [Primjer 9-1](#) pokazuje poboljšani model s nekim dodacima.

Primjer 9-1. app/models.py: model baze podataka uloga

```
class Uloga(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    dozvole = db.Column(db.Integer)
    korisnici = db.relationship('User', backref='role', lazy='dynamic')

    def __init__(self, **kwargs):
        super(Uloga, self).__init__(**kwargs)
        self.permissions = 0

        Ništa: self.permissions = 0
```

Zadano polje je jedan od dodataka ovom modelu. Ovo polje treba postaviti na

Tačno za samo jednu ulogu i Netačno za sve ostale. Uloga označena kao zadana bit će ona koja se dodjeljuje novim korisnicima nakon registracije. Budući da će aplikacija pretraživati tabelu uloga kako bi pronašla zadanu, ova kolona je konfiguirana da ima indeks, jer će to učiniti pretraživanje mnogo brže.

Još jedan dodatak modelu je polje dozvole , koje je cjelobrojna vrijednost koja na kompaktan način definira listu dozvola za ulogu. Budući da će SQLAlchemy ovo polje postaviti na None po defaultu, dodaje se konstruktor klase koji ga postavlja na 0 ako početna vrijednost nije navedena u argumentima konstruktora.

Lista zadataka za koje su potrebne dozvole je očigledno specifična za aplikaciju.

Za Flasky, lista je prikazana u [tabeli 9-1](#).

Tabela 9-1. Dozvole aplikacije

Naziv zadatka	Naziv dozvole	Vrijednost dozvole
Pratite korisnike	PRATITI	1
Komentirajte postove drugih	KOMENTAR	2
Pišite članke	PISATI	4
Moderirajte komentare drugih	MODERATE	8
Pristup administraciji	ADMIN	16

Prednost korištenja ovlasti dvojke za vrijednosti dozvola je u tome što dozvoljava kombiniranje dozvola, dajući svakoj mogućoj kombinaciji dozvola jedinstvenu vrijednost za pohranu u polju dozvola uloge . Na primjer, za korisničku ulogu koja korisnicima daje dozvolu da prate druge korisnike i komentarišu postove, vrijednost dozvole je FOLLOW + COMMENT = 3. Ovo je vrlo efikasan način za pohranjivanje liste dozvola dodijeljenih svakoj ulozi.

Kodni prikaz Tabele 9-1 prikazan je u Primjeru 9-2.

Primjer 9-2. app/models.py: konstante dozvole

Dozvola za razred :

```
PRATI = 1
KOMENTAR = 2
PIŠI = 4
Umjereni = 8
ADMIN = 16
```

Uz postavljene konstante dozvole, nekoliko novih metoda može se dodati modelu uloga za upravljanje dozvolama. Oni su prikazani u primjeru 9-3.

Primjer 9-3. app/models.py: upravljanje dozvolama u Role modelu

```
uloga klase (db.Model):
    # ...

    def add_permission(self, perm):
        ako nije self.has_permission(perm):
            self.permissions += perm

    def remove_permission(self, perm): ako
        self.has_permission(perm):
            self.permissions -= perm

    def reset_permissions(self):
        self.permissions = 0

    def has_permission(self, perm): vrati
        self.permissions & perm == perm
```

Metode add_permission(), remove_permission() i reset_permission() koriste osnovne aritmetičke operacije za ažuriranje liste dozvola. Metoda has_permission() je najkompleksnija od skupa, jer se oslanja na **bitove i operator &** da proverite da li kombinovana vrednost dozvole uključuje datu osnovnu dozvolu. Možete se igrati sa ovim metodama u Python ljudscima:

```
(venv) $ flask shell >>> r
= Uloga(name='Korisnik') >>>
r.add_permission(Permission.FOLLOW) >>>
r.add_permission(Permission.WRITE) >>>
r.has_permission(Permission.PRATITI)
Tačno
>>> r.has_permission(Permission.ADMIN)

Netačno >>> r.reset_permissions()
```

```
>>> r.has_permission(Permission.FOLLOW)
False
```

Tabela 9-2 prikazuje popis korisničkih uloga koje će biti podržane u ovoj aplikaciji, zajedno s kombinacijama dozvola koje definiraju svaku od njih.

Tabela 9-2. Uloge korisnika

Uloga korisnika	Dozvole	Opis
Nema	Nema	Pristup aplikaciji samo za čitanje. Ovo se odnosi na nepoznate korisnike koji nisu prijavljeni.
Korisnik	PRATI, KOMENTARIRAJ, PISATI	Osnovne dozvole za pisanje članaka i komentara i praćenje drugih korisnika. Ovo je zadana postavka za nove korisnike.
Moderator	PRATI, KOMENTARIRAJ, PJŠI , UMJERENO	Dodaje dozvolu za moderiranje komentara drugih korisnika.
Administrator	PRATI, KOMENTARIRAJ, PJŠI , UMJERI, ADMIN	Potpuni pristup, koji uključuje dozvolu za promjenu uloga drugih korisnika.

Ručno dodavanje uloga u bazu podataka je dugotrajno i podložno je greškama, tako da se umjesto toga u klasu uloga može dodati metod klase u tu svrhu, kao što je prikazano u **primjeru 9-4**. Ovo će olakšati ponovno kreiranje ispravnih uloga i dozvola tokom testiranja jedinica i, što je još važnije, na proizvodnom serveru nakon što se aplikacija implementira.

Primjer 9-4. app/models.py: kreiranje uloga u bazi podataka

```
class Role(db.Model): #
    @staticmethod def
    insert_roles():
        roles = {
            'Korisnik': [
                Permission.FOLLOW,
                Permission.COMMENT, Permission.WRITE],
            'Moderator': [Permission.FOLLOW,
                          Permission.COMMENT, Permission.WRITE, Permission.MODERATE],
            'Administrator': [
                Permission.FOLLOW, Permission.COMMENT, Permission.WRITE,
                Permission.MODERATE, Permission.ADMIN],
        }
        default_role = 'Korisnik'
        for r in roles:
            role = Role.query.filter_by(name=r).first()
            if role is None:
                role = Role(name=r)
                role.reset_permissions()
                for perm in roles[r]:
                    role.add_permission(perm)
            role.default = (role.name == default_role)
```

```
db.session.add(role)
db.session.commit()
```

Funkcija `insert_roles()` ne kreira direktno nove objekte uloge. Umjesto toga, pokušava pronaći postojeće uloge po imenu i ažurirati ih. Novi objekt uloge kreira se samo za uloge koje već nisu u bazi podataka. Ovo je učinjeno kako bi se lista uloga mogla ažurirati u budućnosti kada je potrebno izvršiti promjene. Da biste dodali novu ulogu ili promijenili dodjelu dozvola za ulogu, promijenite rječnik uloga na vrhu funkcije i zatim ponovo pokrenite funkciju. Imajte na umu da uloga "Anonimno" ne mora biti predstavljena u bazi podataka, jer je to uloga koja predstavlja korisnike koji nisu poznati i stoga nisu u bazi podataka.

Takođe imajte na umu da je `insert_roles()` statička metoda, posebna vrsta metode koja ne zahteva kreiranje objekta jer se može pozvati direktno u klasi, na primer, kao `Role.insert_roles()`. Statičke metode ne uzimaju `self` argument kao metode instance.

Dodjela uloga

Kada korisnici registriraju račun u aplikaciji, treba im dodijeliti ispravnu ulogu. Za većinu korisnika, uloga dodijeljena u vrijeme registracije bit će uloga "Korisnik", jer je to uloga koja je označena kao zadana. Jedini izuzetak je za administratora, kome treba od početka dodijeliti ulogu "Administrator". Ovaj korisnik je identifikovan po e-mail adresi pohranjenoj u konfiguracionoj varijabli `FLASKY_ADMIN`, tako da čim se ta adresa e-pošte pojavi u zahtjevu za registraciju može joj se dati ispravna uloga. [Primjer 9-5](#) pokazuje kako se to radi u modelu korisnika

konstruktor.

[Primjer 9-5. app/models.py: odreivanje zadane uloge za korisnike](#)

```
class User(UserMixin, db.Model):
    ...
    # def __init__(self, **kwargs):
    #     super(Korisnik, self).__init__(**kwargs) ako
    #     je self.role None:
    #         ako self.email == current_app.config['FLASKY_ADMIN']:
    #             self.role = Role.query.filter_by(name='Administrator').first()
    #         ako je self.role None:
    #             self.role = Role.query.filter_by(default=True).first()
    # ...

```

Korisnički konstruktor prvo poziva konstruktore osnovnih klasa, a ako nakon toga objekt nema definiranu ulogu, postavlja administratorsku ili zadalu ulogu ovisno o adresi e-pošte .

Verifikacija uloge

Da bi se pojednostavila implementacija uloga i dozvola, modelu korisnika može se dodati pomoćna metoda koja provjerava da li korisnici imaju datu dozvolu u ulozi koja im je dodijeljena. Implementacija se jednostavno oslanja na prethodno dodane metode uloga. To je prikazano u [primjeru 9-6](#).

Primjer 9-6. app/models.py: procjena da li korisnik ima datu dozvolu

```
from flask_login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, perm): vrati
        self.role nije None i self.role.has_permission(perm)

    def is_administrator(self): vrati
        self.can(Permission.ADMIN)

klasa AnonymousUser(AnonymousUserMixin): def
    can(self, permissions): return False

    def is_administrator(self): vrati
        False

login_manager.anonymous_user = Anonimni korisnik
```

Metoda can() dodana modelu korisnika vraća True ako je tražena dozvola prisutna u ulozi, što znači da korisniku treba dozvoliti da izvrši traženi zadatak. Provjera administrativnih dozvola je toliko uobičajena da se također implementira kao samostalna metoda is_administrator() .

Za dodatnu pogodnost kreirana je i prilagođena klasa AnonymousUser koja implementira metode can() i is_administrator() . Ovo će omogućiti aplikaciji da slobodno poziva current_user.can() i current_user.is_administrator() bez potrebe da prvo provjerava da li je korisnik prijavljen. Flask-Login-u je rečeno da koristi prilagođenog anonimnog korisnika aplikacije postavljanjem svoje klase u atribut login_manager.anonymous_user .

Za slučajeve u kojima cijela funkcija pogleda treba biti dostupna samo korisnicima s određenim dozvolama, može se koristiti prilagođeni dekorater. [Primjer 9-7](#) pokazuje implementaciju dva dekoratora, jedan za generičke provjere dozvola i jedan koji posebno provjerava administratorsku dozvolu.

Primjer 9-7. app/decorators.py: prilagođeni dekorateri koji provjeravaju korisničke dozvole

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import permission

def permission_required(permission):
    def dekorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)

        return decorated_function
    return dekorator

def admin_required(f):
    return permission_required(Permission.ADMIN)
```

Ovi dekorateri su napravljeni uz pomoć **funkcijskih alata** paket iz Python standardne biblioteke i vrati odgovor 403, "Zabranjeni" HTTP statusni kod, kada trenutni korisnik nema traženu dozvolu. U poglavlju 3 kreirane su prilagođene stranice grešaka za greške 404 i 500, tako da se sada stranica za grešku 403 dodaje na sličan način.

Slijede dva primjera koji pokazuju upotrebu ovih dekoratera:

```
iz .decorators import admin_required, permission_required

@login_required
@main.route('/admin')
@admin_required
def for_admins_only():
    return "Za administratore!"

@login_required
@main.route('/moderate')
@permission_required(Permission.MODERATE)
def for_moderators_only():
    return "Za moderatore komentara!"
```

Kao pravilo, dekorator rute iz Flask-a treba biti dat prvi kada se koristi više dekoratora u funkciji pogleda. Preostale dekoratere treba dati redoslijedom kojim trebaju procijeniti kada se pozove funkcija pogleda. U ova dva slučaja, prvo treba provjeriti stanje autentifikacije korisnika, budući da korisnik mora biti preusmjeren na prompt za prijavu ako se utvrdi da nije autenticiran.

Dozvole će se možda morati provjeriti i iz predložaka, tako da klasa Permission sa svim svojim konstantama treba da im bude dostupna. Da biste izbjegli dodavanje tem-

Argument ploče u svakom pozivu render_template() , može se koristiti kontekstualni procesor. Procesori konteksta čine varijable dostupnim svim šablonima tokom renderovanja. Ova promjena je prikazana u [primjeru 9-8](#).

Primjer 9-8. app/main/__init__.py: dodavanje klase Permission u kontekst predloška

```
@main.app_context_processor
def inject_permissions(): vrati
    dict(Permission=Dozvola)
```

Nove uloge i dozvole se mogu koristiti u jediničnim testovima. [Primjer 9-9](#) prikazuje dva testa. Izvorni kod na GitHubu uključuje po jedan za svaku ulogu.

Primjer 9-9. tests/test_user_model.py: jedinični testovi za uloge i dozvole

```
klasa UserModelTestCase(unittest.TestCase):
    # ...

    def test_user_role(self):
        u = Korisnik(email='john@example.com', lozinka='mačka')
        self.assertTrue(u.can(Permission.FOLLOW))
        self.assertTrue(u.can(Permission.COMMENT))
        self.assertTrue( u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))

    def test_anonymous_user(self): u =
        AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))
        self.assertFalse(u.can(Permission.COMMENT))
        self.assertFalse(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 9a da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

Prije nego što pređete na sljedeće poglavlje, dodajte nove uloge vašoj razvojnoj bazi podataka u sesiji Ijuske:

```
(venv) $ flask shell >>>
Role.insert_roles()
>>> Role.query.all()
[<Uloga 'Administrator'>, <Uloga 'Korisnik'>, <Uloga 'Moderator'>]
```

Također je dobra ideja ažurirati popis korisnika tako da svi korisnički nalozi koji su kreirani prije postojanja uloga i dozvola imaju dodijeljenu ulogu. Možete pokrenuti sljedeći kod u Python ljudi da izvršite ovo ažuriranje:

Korisnički sistem je sada prilično kompletan. Sljedeće poglavlje će ga koristiti za kreiranje stranica korisničkog profila.

User Proles

U ovom poglavlju implementirani su korisnički profili za Flasky. Sve društveno osvještene stranice svojim korisnicima daju stranicu profila, na kojoj se prikazuje sažetak učešća korisnika na web stranici. Korisnici mogu reklamirati svoje prisustvo na web stranici dijeljenjem URL-a na svojoj stranici profila, stoga je važno da URL-ovi budu kratki i lako pamtljivi.

Prole Information

Kako bi stranice korisničkih profila bile zanimljivije, neke dodatne informacije o korisnicima mogu se pohraniti u bazu podataka. U [primjeru 10-1](#) model korisnika je proširen sa nekoliko novih polja.

Primjer 10-1. app/models.py: polja informacija o korisniku

```
class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    about_me = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow) last_seen
    = db.Column(db.DateTime(), default=datetime.utcnow)
```

Nova polja pohranjuju pravo ime korisnika, lokaciju, vlastitu biografiju, datum registracije i datum posljednje posjete. Polju about_me je dodijeljen tip db.Text(). Razlika između db.String i db.Text je u tome što je db.Text polje promjenjive dužine i kao takvo ne treba maksimalnu dužinu.

Dve vremenske oznake dobijaju podrazumevanu vrednost trenutnog vremena. Imajte na umu da datetime.utcnow nedostaje () na kraju. To je zato što zadani argument u db.Column() može uzeti funkciju kao vrijednost. Svaki put mora biti zadana vrijednost

generirano, SQLAlchemy poziva funkciju da je proizvede. Ova zadana vrijednost je sve što je potrebno za upravljanje poljem member_since .

Polje last_seen je također inicijalizirano trenutnim vremenom nakon kreiranja, ali ga treba osvježiti svaki put kada korisnik pristupi stranici. Za izvođenje ovog ažuriranja može se dodati metoda u klasu User . To je prikazano u [primjeru 10-2](#).

Primjer 10-2. app/models.py: osvježavanje vremena posljednje posjete korisnika

```
class User(UserMixin, db.Model):
    # ...

    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self) db.session.commit()
```

Da bi datum posljednje posjete za sve korisnike bio ažuriran, metoda ping() mora biti pozvana svaki put kada se primi zahtjev od korisnika. Budući da se obrađivač before_app_request u auth nacrtu pokreće prije svakog zahtjeva, to može učiniti lako, kao što je prikazano u [primjeru 10-3](#).

Primjer 10-3. app/auth/views.py: pingovanje prijavljenog korisnika

```
@auth.before_app_request
def before_request(): ako
    current_user.is_authenticated:
        current_user.ping() ako nije
        current_user.confirmed \ i
            request.endpoint \ and
            request.blueprint != 'auth' \ and
            request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))
```

Stranica korisnika Prole

Kreiranje stranice profila za svakog korisnika ne predstavlja nikakve nove izazove.

[Primjer 10-4](#) prikazuje definiciju rute.

Primjer 10-4. app/main/views.py: ruta prole stranice

```
@main.route('/user/<username>') def
user(username): user =
    User.query.filter_by(username=username).first_or_404() return
    render_template('user.html', user=user)
```

Ova ruta je dodata u glavni plan. Za korisnika po imenu john, stranica profila će biti na `http://localhost:5000/user/john`. Korisničko ime dato u URL-u se pretražuje u bazi podataka i, ako se pronađe, predložak `user.html` se prikazuje s njim kao argumentom. Nevažeće korisničko ime poslano na ovu rutu će uzrokovati vraćanje greške 404. Sa Flask-`SQLAlchemy`, slučajevi pretraživanja i greške mogu se lijepo kombinirati u jednoj naredbi koristeći `first_or_404()` metodu objekta upita. Predložak `user.html` će morati da predstavi informacije o korisniku, tako da prima korisnički objekat kao argument. Početna verzija ovog predloška prikazana je u [primjeru 10-5.](#)

Primjer 10-5. `app/templates/user.html`: predložak korisničkog prolea

```
{% proširuje "base.html" %} {%
block title %}Flasky - {{ user.username }}{% endblock %}

{% block page_content %} <div
class="page-header">
<h1>{{ user.username }}</h1> {% if
user.name ili user.location %}
<p>
    {% if user.name %}{{ user.name }}{% endif %} {% if user.location
%}
        Sa <a href="http://maps.google.com/?q={{ user.location }}">
            {{ user.location }} </a> {% if
        endif %} </p> {% endif %} {% if
        current_user.is_administrator() %}
<p><a href="mailto:
{{ user.email }}">{{ user.email }}</a></p>
    {% endif %} {% if user.about_me %}
<p>{{ user.about_me }}</p>{% endif %}
</p>{%
endif %}

<p>
    Član od {{ moment(user.member_since).format('L') }}.
    Posljednji put viden {{ moment(user.last_seen).fromNow() }}. </p> </
div> {% endblock %}
```

Ovaj predložak ima nekoliko zanimljivih detalja implementacije:

- Polja imena i lokacije prikazuju se unutar jednog elementa `<p>`. Jinja2 uslov osigurava da se element `<p>` kreira samo kada je barem jedno od polja definirano.
- Polje korisničke lokacije prikazuje se kao link na upit Google Maps, tako da se klikom na njega otvara mapa centrirana na lokaciji.

- Ako je prijavljen korisnik administrator, tada se prikazuje adresa e-pošte korisnika, prikazana kao mailto link. Ovo je korisno kada administrator pregleda stranicu profila drugog korisnika i treba da kontaktira korisnika.
- Dvije vremenske oznake za korisnika se prikazuju na stranici koristeći Flask-Moment, kao što je prikazano u poglaviju 3.

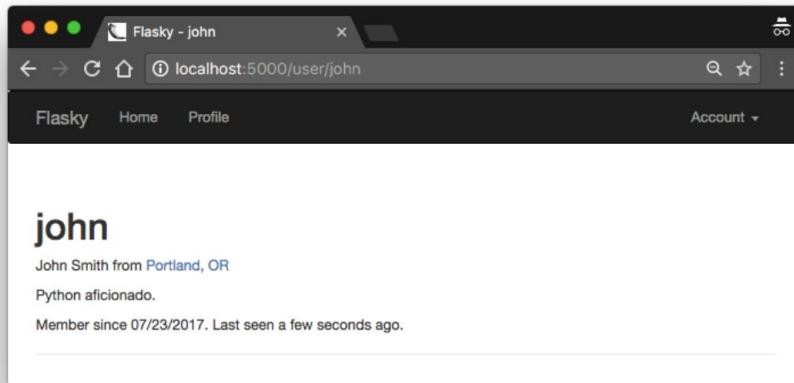
Kako će većina korisnika htjeti lak pristup svojoj stranici profila, link ka njoj može se dodati na navigacijsku traku. Relevantne promjene u predlošku base.html prikazane su u [primjeru 10-6](#).

Primjer 10-6. app/templates/base.html: dodajte link na prole stranicu u navigacijskoj traci

```
{% if current_user.is_authenticated %} <li>
<a href="{{ url_for('main.user',
                      username=current_user.username) }}">
    Profil </
<a> </li> {%
endif %}
```

Korištenje uvjeta za vezu stranice profila je neophodno jer se navigacijska traka također prikazuje za neovlaštene korisnike, u kom slučaju se veza profila preskače.

[Slika 10-1](#) prikazuje kako stranica profila izgleda u pretraživaču. Takođe je prikazan i novi link profila u traci za navigaciju.



Slika 10-1. Stranica korisnika



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 10a da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

Prole Editor

Postoje dva različita slučaja upotrebe vezana za uređivanje korisničkih profila. Najočiglednije je da korisnici moraju imati pristup stranici na kojoj mogu unijeti informacije o sebi koje će predstaviti na svojim stranicama profila. Manje očigledan, ali takođe važan uslov je da se administratorima dozvoli da uređuju profile drugih korisnika – ne samo njihove lične informacije, već i druga polja u modelu korisnika kojima korisnici nemaju direktni pristup, kao što je uloga korisnika. Budući da se dva zahtjeva za uređivanje profila bitno razlikuju, kreirat će se dva različita obrasca.

Prole uređivač na korisničkom nivou

Obrazac za uređivanje profila za redovne korisnike prikazan je u [primjeru 10-7](#).

Primjer 10-7. app/main/forms.py: uredi prole obrazac

```
class EditProfileForm(FlaskForm): name  
    = StringField('Pravo ime', validators=[Length(0, 64)]) location =  
    StringField('Location', validators=[Length(0, 64)]) about_me = TextAreaField('  
O meni') submit = SubmitField('Submit')
```

Imajte na umu da su sva polja u ovom obrascu opcionala, validator dužine dozvoljava dužinu od nule kao minimum. Definicija rute koja koristi ovaj obrazac prikazana je u [primjeru 10-8](#).

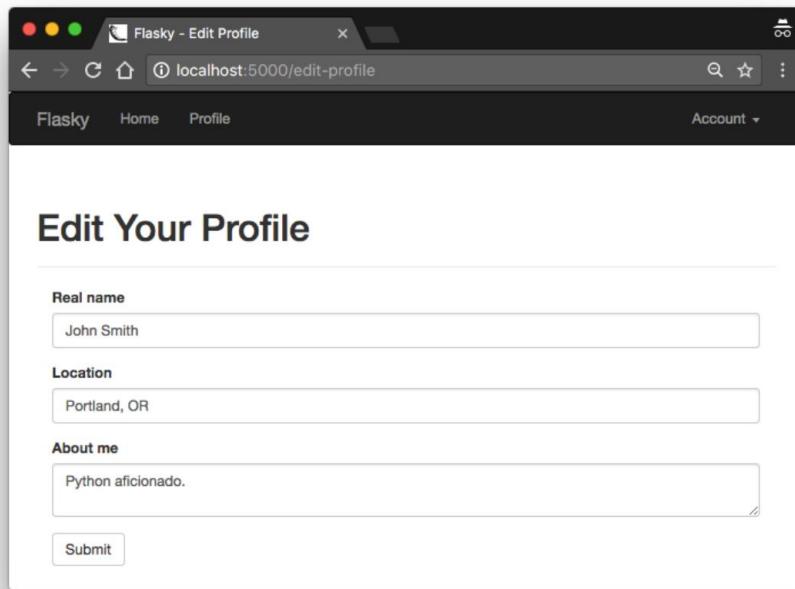
Primjer 10-8. app/main/views.py: uredi prole rutu

```
@main.route('/edit-profile', methods=['GET', 'POST']) @login_required  
def edit_profile(): form = EditProfileForm() if form.validate_on_submit():  
    current_user.name = form.name.data current_user.location =  
        form.location.data current_user.about_me =  
            form.about_me.data  
            db.session.add(current_user._get_current_object())  
            db.session.commit() flash('Vaš profil je ažuriran.') return  
            redirect(url_for('.user', username=current_user.username))  
  
form.name.data = trenutnog_user.name
```

```
form.location.data = current_user.location  
form.about_me.data = current_user.about_me  
return  
render_template('edit_profile.html', form=form)
```

Kao i prethodnim obrascima, podaci povezani sa svakim poljem obrasca dostupni su na form.<naziv-polja>.data. Ovo je korisno ne samo za dobivanje vrijednosti koje je korisnik dostavio, već i za pružanje početnih vrijednosti koje se prikazuju korisniku za uređivanje. Kada je form.validate_on_submit() False, tri polja u ovom obrascu se inicijaliziraju iz odgovarajućih polja u current_user. Zatim, kada se obrazac posalje, atributi podataka polja obrasca sadrže ažurirane vrijednosti, tako da se one vraćaju u polja korisničkog objekta prije nego što se objekt pohrani natrag u bazu podataka.

Slika 10-2 prikazuje stranicu za uređivanje profila.



Slika 10-2. Prole editor

Kako bi korisnicima olakšali da dođu do ove stranice, može se dodati direktni link na stranicu profila, kao što je prikazano u [primjeru 10-9](#).

Primjer 10-9. app/templates/user.html: uređi prole link

```
{% if user == trenutni_user %} <a
class="btn btn-default" href="{{ url_for('.edit_profile') }}"> Uredi profil </a> {%
endif %}
```

Uslov koji obuhvata vezu učinit će da se link pojavljuje samo kada korisnici gledaju svoje profile.

Prole uređivač na nivou administratora

Obrazac za uređivanje profila za administratore je složeniji od onog za obične korisnike. Pored tri polja sa informacijama o profilu, ovaj obrazac omogućava administratorima da uređuju korisničku e-poštu, korisničko ime, potvrđen status i ulogu. Obrazac je prikazan u **primjeru 10-10.**

Primjer 10-10. app/main/forms.py: prole obrazac za uređivanje za administratore

```
klasa EditProfileAdminForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    korisničko ime = StringField('Username',
                                 validators=[ DataRequired(), Length(1, 64), Regexp('^[A-
Za-z][A-Za-z0-9_]*$', 0,
                                         'Korisnička imena moraju imati samo slova, brojeve, tačke ili
                                         'podvlake'')])
    potvrđeno = BooleanField('Confirmed') role =
    SelectField('Role', coerce=int) name = StringField('Pravo ime', validators=[Length]
(0, 64)) location = StringField('Location', validators=[Length(0, 64)]) about_me
= TextAreaField('O meni') submit = SubmitField('Submit')

    def __init__(self, korisnik, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs) self.role.choices
        = [(role.id, role.name) za ulogu u Role.query.order_by(Role.name).all()]

        self.user = korisnik

    def validate_email(self, field):
        if field.data != self.user.email \ User.query.filter_by(email=field.data).first(): raise
        ValidationError('E-pošta je već registrovana.')

    def validate_username(self, field):
        if field.data != self.user.username \
```

```
User.query.filter_by(username=field.data).first():
    raise ValidationError('Korisničko ime je već u upotrebi.')
```

SelectField je WTFormov omot za < select> HTML kontrolu obrasca, koja implementira padajuću listu, koja se koristi u ovom obrascu za odabir korisničke uloge. Instanca SelectField mora imati stavke postavljene u atributu izbora . Moraju se dati kao lista torki, pri čemu se svaki tuple sastoji od dvije vrijednosti: identifikatora za stavku i teksta koji će se prikazati u kontroli kao string. Lista izbora je postavljena u konstruktoru obrasca, sa vrednostima dobijenim iz modela uloga sa upitom koji sve uloge sortira po abecednom redu po imenu. Identifikator za svaki tuple je postavljen na id svake uloge, a pošto su to cijeli brojevi, argument coerce =int se dodaje u SelectField konstruktor tako da se vrijednosti polja pohranjuju kao cijeli brojevi umjesto zadanih, a to su stringovi.

Polja e-pošte i korisničkog imena su konstruisana na isti način kao u obrascima za autentifikaciju, ali njihova validacija zahteva pažljivo rukovanje. Uslov validacije koji se koristi za oba ova polja mora prvo da proveri da li je izvršena promena u polju, a tek kada dođe do promene treba da obezbedi da nova vrednost ne duplira vrednost drugog korisnika. Kada se ova polja ne mijenjaju, valjanost bi trebala proći. Da bi implementirao ovu logiku, konstruktor obrasca prima korisnički objekat kao argument i sprema ga kao varijablu člana, koja se kasnije koristi u prilagođenim metodama validacije.

Definicija rute za editor profila administratora prikazana je u [primjeru 10-11](#).

Primjer 10-11. app/main/views.py: uredi rutu prole za administratore

```
iz ..decorators import admin_required

@admin_required
def edit_profile_admin(id): korisnik = User.query.get_or_404(id)
form = EditProfileAdminForm(user=user) if form.validate_on_submit():

    user.email = form.email.data
    user.username = form.username.data
    user.confirmed = form.confirmed.data
    user.role = Uloga.query.get(form.role.data)
    user.name = form.name.data
    user.location =
    form.location.data
    user.about_me =
    form.about_me.data
    db.session.add(user)
    db.session.commit() flash(' Profil je ažuriran.') return
    redirect(url_for('.user', korisničko_ime=user.username))
```

```
form.email.data = user.email
```

```

form.username.data = user.username
form.confirmed.data = user.confirmed
form.role.data = user.role_id
form.name.data = user.name
form.location.data = user.location
form.about_me.data = user.about_me
return render_template('edit_profile.html',
form=form, user=user)

```

Ova ruta ima uglavnom istu strukturu kao i jednostavnija za obične korisnike, ali uključuje dekorator `admin_required` kreiran u [Poglavlju 9](#), koji će automatski vratiti grešku 403 za sve korisnike koji nisu administratori koji pokušaju da koriste ovu rutu.

Korisnički id je dat kao dinamički argument u URL-u, tako da se Flask-SQLAlchemy-jeva `get_or_404()` funkcija može koristiti, znajući da ako je id nevažeći zahtjev će vratiti grešku koda 404. `SelectField` koje se koristi za ulogu korisnika također zaslužuje da se prouči. Prilikom postavljanja početne vrijednosti za polje, `role_id` se dodjeljuje `field.role.data` jer lista torki postavljenih u atributu izbora koristi numeričke identifikatore da referencira svaku opciju. Kada se obrazac pošalje, id se izdvaja iz atributa podataka polja i koristi u upitu za ponovno učitavanje odabranog objekta uloge po njegovom ID -u. Argument `coerce =int` koji se koristi u deklaraciji `SelectField` u obrascu osigurava da se atribut podataka ovog polja uvijek konvertuje u cijeli broj.

Za povezivanje na ovu stranicu, na stranicu korisničkog profila dodaje se još jedno dugme, kao što je prikazano u [primjeru 10-12](#).

Primjer 10-12. `app/templates/user.html`: prole veza za uređivanje za administratore

```

{% if current_user.is_administrator() %} <a
class="btn btn-danger"
href="{{ url_for('.edit_profile_admin', id=user.id) }}> Uredi profil
[Admin] </a> {% endif %}

```

Ovo dugme je prikazano sa drugačijim Bootstrap stilom da skrene pažnju na njega. Uslov koji ga obavlja čini da se dugme pojavljuje na stranicama profila samo ako prijavljeni korisnik ima ulogu administratora.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 10b da provjerite ovu verziju aplikacije.

User Avatars

Izgled stranica profila može se poboljšati prikazivanjem slika avatara korisnika. U ovom dijelu ćete naučiti kako dodati avatare korisnika koje pruža **Gravatar**, vodeći servis avatara. Gravatar povezuje slike avatara s adresama e-pošte. Korisnici kreiraju nalog na <https://gravatar.com> a zatim otpremite njihove slike. Usluga izlaže avatar korisnika putem posebno kreiranog URL-a koji uključuje MD5 hash adresu e-pošte korisnika, koji se može izračunati na sljedeći način:

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

URL-ovi avatara se zatim generišu dodavanjem MD5 heša na <https://secure.gravatar.com/avatar/> URL. Na primjer, možete upisati <https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6> u adresnu traku vašeg pretraživača da dobijete sliku avatara za adresu e-pošte john@example.com, ili zadanu sliku avatara ako je ta adresa e-pošte nema registrovan avatar. Nakon što napravite osnovni URL avatara, nekoliko argumenata stringa upita može se koristiti za konfigurisanje karakteristika slike avatara, kao što je opisano u [tabeli 10-1](#).

Tabela 10-1. Argumenti niza upita Gravatar

Argument ime	Opis
s	Veličina slike, u pikselima.
r	Ocjena slike. Opcije su "g", "pg", "r" i "x".
d	Zadani generator slika za korisnike koji nemaju registrovane avatare na usluzi Gravatar. Opcije su "404" za vraćanje greške 404, URL koji upućuje na zadanu sliku ili jedan od sljedećih generatora slika: "mm", "identicon", "monsterid", "wavatar", "retro" ili "prazno".
fd	Prisilite upotrebu zadanih avatara.

Na primjer, dodavanjem ?d=identicon URL-u avatara za john@example.com generirat će se drugačiji zadani avatar koji je baziran na geometrijskom dizajnu. Sve ove opcije za generiranje URL-ova avatara mogu se dodati u korisnički model. Implementacija je prikazana u [primjeru 10-13](#).

Primjer 10-13. app/models.py: gravatar generiranje URL-a

```
uvoz hashlib iz
zahtjeva za uvoz flask

class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
```

```

url = 'https://secure.gravatar.com/avatar' hash =
hashlib.md5(self.email.lower().encode('utf-8')).hexdigest() return '{url}/{hash }?
s={size}&d={default}&r={rating}'.format(
    url=url, hash=hash, veličina=veličina, default=default, rating=ocjen)

```

URL avatara se generira iz osnovnog URL-a, MD5 heša korisničke adrese e-pošte i argumenata, od kojih svi imaju zadane vrijednosti. Imajte na umu da je jedan od zahtjeva usluge Gravatar da adresa e-pošte sa koje se dobija MD5 hash bude normalizirana da sadrži samo mala slova abecednih znakova, tako da se i konverzija dodaje ovoj metodi. Sa ovom implementacijom lako je generirati URL-ove avatara u Ijusci Python:

```

(venv) $ flask shell >>> u
= User(email='john@example.com') >>>
u.gravatar() 'https://secure.gravatar.com/
avatar/d4c74594d841139328695756648b6bd6?s=100&d=identicon&r=g' >>> u.gravatar(size=256)
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=256&d=
identicon&r=g'

```

Metoda `gravatar()` se također može pozvati iz Jinja2 šablona. [Primjer 10-14](#) pokazuje kako se avatar od 256 piksela može dodati na stranicu profila.

Primjer 10-14. app/templates/user.html: dodavanje avatara na prole stranicu

```

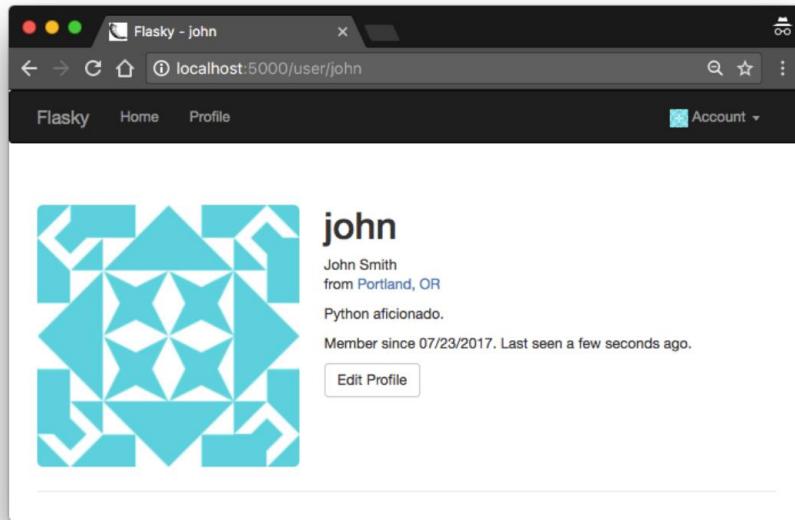
...

...
</div>
...

```

CSS klasa profilne sličice pomaže pri pozicioniranju slike na stranici. Element `<div>` koji slijedi nakon slike inkapsulira informacije o profilu i koristi CSS klasu zaglavljiva profila za poboljšanje formatiranja. Možete vidjeti definiciju CSS klase u GitHub spremištu za aplikaciju.

Koristeći sličan pristup, osnovni predložak dodaje malu sličicu prijavljenog korisnika u navigacijsku traku. Za bolje formatiranje slika avatara na stranici, koriste se prilagođene CSS klase. Možete ih pronaći u spremištu izvornog koda u datoteci `styles.css` dodanoj u fasciklu statičkog fajla aplikacije i na koju se upućuje iz `base.html` šablona. [Slika 10-3](#) prikazuje stranicu korisničkog profila sa avataram.



Slika 10-3. Korisnička prole stranica sa avatarom



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 10c da provjerite ovu verziju aplikacije.

Generisanje avatara zahteva generisanje MD5 heša, što zahteva CPU intenzivnu operaciju. Ako za stranicu treba generirati veliki broj avatara, onda se računski rad može zbrajati i postati značajan. Budući da će MD5 hash za korisnika ostati konstantan sve dok adresa e-pošte ostane ista, može se keširati u modelu korisnika. [Primjer 10-15](#) pokazuje promjene u modelu korisnika za pohranjivanje MD5 heševa u bazu podataka.

[Primjer 10-15. app/models.py: gravatar generiranje URL-a sa keširanjem MD5 heševa](#)

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
        # ...
        if self.email is not None and self.avatar_hash is None:
            self.avatar_hash = generate_gravatar(self.email)
```

```

    self.avatar_hash = self.gravatar_hash()

def change_email(self, token):
    ...
    # self.email = new_email
    self.avatar_hash = self.gravatar_hash()
    db.session.add(self)
    return Tačno

def gravatar_hash(self):
    vrati hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()

def gravatar(self, size=100, default='identicon', rating='g'):
    if request.is_secure(): url =
        'https://secure.gravatar.com/avatar' else: url = 'http://
    www.gravatar.com/avatar' hash = self.avatar_hash ili
        self.gravatar_hash() return '{url}/{hash}?s={size}
&d={default}&r={rating}'.format(
    url=url, hash=hash, veličina=veličina, default=default, rating=ocjen)

```

Kako bi se izbjeglo duplicitiranje logike za izračunavanje gravatar hash-a, dodaje se nova metoda, gravatar_hash(), koja obavlja ovaj zadatak. Tokom inicijalizacije modela, heš se pohranjuje u novu kolonu modela avatar_hash . Ako korisnik ažurira adresu e-pošte, heš se ponovo izračunava. Metoda gravatar() koristi pohranjeni hash ako je dostupan, a ako nije, generira novi hash kao i prije.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 10d da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

U sljedećem poglavlju bit će kreiran mehanizam za blogovanje koji pokreće ovu aplikaciju.

Blog Postovi

Ovo poglavlje je posvećeno implementaciji glavne funkcije Flaskyja, a to je omogućavanje korisnicima da čitaju i pišu postove na blogu. Ovdje ćete naučiti nekoliko novih tehnika za ponovnu upotrebu predložaka, paginaciju dugih lista stavki i rad s obogaćenim tekstom.

Slanje i prikaz objava na blogu

Da bi se podržali postovi na blogu, neophodan je novi model baze podataka koji ih predstavlja. Ovaj model je prikazan u [primjeru 11-1](#).

Primjer 11-1. app/models.py: Post model

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    ...
    # postova = db.relationship('Post', backref='author', lazy='dynamic')
```

Objava na blogu je predstavljena tijelom, vremenskom oznakom i odnosom jedan-prema-više iz modela korisnika . Polje tijela definirano je tipom db.Text tako da nema ograničenja u dužini.

Obrazac koji će biti prikazan na glavnoj stranici aplikacije omogućava korisnicima da napišu blog post. Ovaj oblik je vrlo jednostavan; sadrži samo tekstualno područje u koje se može upisati blog post i dugme za slanje. Definicija obrasca je prikazana u [primjeru 11-2](#).

Primjer 11-2. app/main/forms.py: obrazac za objavu na blogu

```
class PostForm(FlaskForm):
    body = TextAreaField("Šta ti je na umu?", validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Funkcija prikaza index() rukuje formom i proslijeđuje listu starih blog postova u šablon, kao što je prikazano u [primjeru 11-3](#).

Primjer 11-3. app/main/views.py: ruta početne stranice sa objavom na blogu

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = PostForm()
    if current_user.can(Permission.WRITE_ARTICLES) and
       form.validate_on_submit():
        post = Post()
        post.body = form.body.data
        post.autor = current_user._get_current_object()
        db.session.add(post)
        db.session.commit()
        return redirect(url_for('.index'))
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', form=form, posts=posts)
```

Ova funkcija pregleda proslijeđuje obrazac i kompletну listu blog postova u predložak.

Lista postova je poređana prema vremenskoj oznaci, u opadajućem redoslijedu. Obrazac za objavu na blogu se obrađuje na uobičajen način, uz kreiranje nove instance Posta kada se primi valjana prijava. Dozvola trenutnog korisnika za pisanje članaka se provjerava prije nego što se dozvoli nova objava.

Obratite pažnju kako je atribut autora novog objekta posta postavljen na izraz `current_user._get_current_object()`. Varijabla `current_user` iz Flask Login, kao i sve varijable konteksta, implementirana je kao proxy objekt lokalnog niti. Ovaj objekat se ponaša kao korisnički objekat, ali je zapravo tanak omotač koji sadrži stvarni korisnički objekat unutra. Bazi podataka je potreban pravi korisnički objekat, koji se dobija pozivom `_get_current_object()` na proxy objektu.

Obrazac je prikazan ispod pozdrava u predlošku `index.html`, nakon čega slijede postovi na blogu. Lista postova na blogu je prvi pokušaj da se kreira vremenska linija blog postova, sa svim blog postovima u bazi podataka navedenim hronološkim redom od najnovijih do najstarijih.

Promjene u šablonu su prikazane u [primjeru 11-4](#).

Primjer 11-4. app/templates/index.html: šablon početne stranice sa objavama na blogu

```
{% proširuje "base.html" %} {%
import "bootstrap/wtf.html" kao wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }} {% endif %} </div> <ul class="posts">
    {% za objavu u objavama % } <li class="post"> <div class="profile-
    thumbnail">

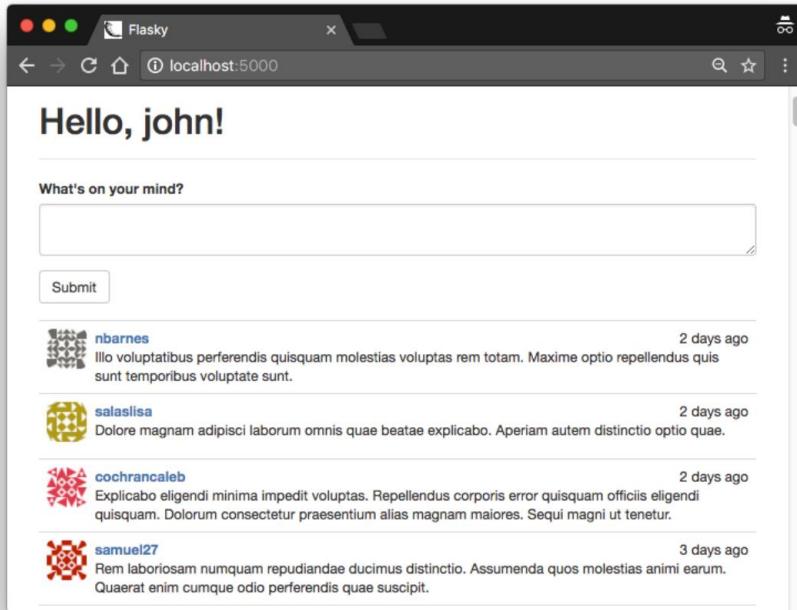
        <a href="{{ url_for('.user', korisničko ime=post.author.username) }}>
            {{ moment(post.timestamp).fromNow() }}</div> <div
    class="post-author">
        <a href="{{ url_for('.user', korisničko ime=post.author.username) }}>
            {{ post.author.username }} </a> </div> <div class="post- body">{{ post.body }}</div>
    </li> {% endfor %} </ul>
    ...

```

Imajte na umu da se metoda User.can() koristi za preskakanje obrasca objave na blogu za korisnike koji nemaju dozvolu WRITE u svojoj ulozi. Lista blog postova implementirana je kao HTML neuređena lista, sa CSS klasama koje joj daju ljepše formatiranje. Mali avatar autora prikazan je na lijevoj strani, a i avatar i korisničko ime autora prikazani su kao veze do stranice profila korisnika. Korišteni CSS stilovi pohranjeni su u datoteci styles.css koja se nalazi u statičkom direktoriju aplikacije. Ovu datoteku možete pregledati u GitHub spremištu. **Slika 11-1** prikazuje početnu stranicu sa obrascem za podnošenje i listom postova na blogu.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11a da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.



Slika 11-1. Početna stranica sa formularom za prijavu bloga i listom blog postova

Blog Postovi na Prole stranicama

Stranica korisničkog profila može se poboljšati prikazivanjem liste blog postova čiji je autor korisnik.

Primjer 11-5 pokazuje promjene u funkciji prikaza da bi se dobila lista objava.

Primjer 11-5. app/main/views.py: ruta prole stranice sa objavama na blogu

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first() ako je korisnik
    Ništa: abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

Lista blog postova za korisnika se dobija iz odnosa User.posts . Ovo radi kao objekat upita, tako da se filteri kao što je order_by() mogu koristiti na njemu kao u običnom objektu upita.

Predložak user.html mora imati isto HTML stablo koje prikazuje listu blog postova u index.html, ali potreba za održavanjem dvije identične kopije dijela HTML koda nije idealna. Za slučajeve kao što je ovaj, Jinja2 direktiva uključivanja je vrlo korisna.

Isječak HTML-a koji generiše listu postova može se premjestiti u zasebnu datoteku koju mogu uključiti i index.html i user.html. **Primjer 11-6** pokazuje kako ovo uključuje izgleda u user.html.

Primjer 11-6. app/templates/user.html: predložak prole stranice sa objavama na blogu

```
...
<h3>Objave od {{ user.username }}</h3> {%
include '_posts.html' %}
...
...
```

Da bi se dovršila ova reorganizacija, stablo iz index.html se premješta u novi predložak _posts.html i zamjenjuje drugom direktivom uključivanja kao što je upravo prikazana. Imajte na umu da upotreba prefiksa donje crte u nazivu predloška _posts.html nije uslov; ovo je samo konvencija za razlikovanje potpunih i djelomičnih šablona.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11b da provjerite ovu verziju aplikacije.

Paginiranje dugih lista postova na blogu

Kako stranica bude rasla i broj postova na blogu se povećavao, postat će sporo i nepraktično prikazivati kompletну listu postova na početnoj stranici i stranicama profila. Velikim stranicama je potrebno više vremena za generiranje, preuzimanje i prikazivanje u web pretraživaču, tako da se kvalitet korisničkog iskustva smanjuje kako stranice postaju veće. Rješenje je paginirati podatke i prikazati ih u komadima.

Kreiranje lažnih podataka blog postova

Da biste mogli raditi sa više stranica blog postova, potrebno je imati testnu bazu podataka sa velikom količinom podataka. Ručno dodavanje novih unosa u bazu podataka je dugotrajno i zamorno; automatizovano rešenje je prikladnije. Postoji nekoliko Python paketa koji se mogu koristiti za generiranje lažnih informacija. Prilično kompletan je Faker, koji se instalira sa pip-om:

```
(venv) $ pip install faker
```

Paket Faker nije, striktno govoreći, zavisnost aplikacije, jer je potreban samo tokom razvoja. Da bi se odvojile proizvodne zavisnosti od razvojnih, datoteka requirements.txt može se zamijeniti poddirektorijumom zahtjeva koji pohranjuje različite skupove ovisnosti. Unutar ovog novog poddirektorijuma, datoteka dev.txt može navesti zavisnosti koje su neophodne za razvoj, a prod.txt datoteka može navesti zavisnosti koje su potrebne u proizvodnji. Pošto postoji veliki broj zavisnosti koje će biti u obe liste, za njih se dodaje common.txt fajl, a zatim liste dev.txt i prod.txt koriste prefiks -r da ga uključe.

Primjer 11-7 prikazuje datoteku dev.txt.

Primjer 11-7. Zahtevi/dev.txt: razvojni zahtevi le

```
-r common.txt
faker==0.7.18
```

Primer 11-8 pokazuje novi modul koji je dodat aplikaciji koji sadrži dve funkcije koje generišu lažne korisnike i objave.

Primjer 11-8. app/fake.py: generiranje lažnih korisnika i blog postova

```
iz slučajnog uvoza randint iz
sqlalchemy.exc import IntegrityError iz faker import
Faker iz . import db from .models import User, Post
```

```
def users(count=100):
    fake = Lažni () i = 0
    dok i < count: u =
        Korisnik(email=fake.email(), korisničko
                  ime=fake.user_name(),
                  lozinka='password',
                  potvrđeno=Tačno ,
                  name=lažno.ime(),
                  lokacija=lažni.grad(),
                  about_me=lažni.text(),
                  member_since=fake.past_date())
        db.session.add(u) pokušajte: db.session.commit()

        i += 1
    osim IntegrityError:
        db.session.rollback()

def postovi(count=100):
    lažni = Lažni ()
    user_count = User.query.count()
```

```
za i u rasponu (broj):
    u = User.query.offset(randint(0, user_count - 1)).first() p =
        Post(body=fake.text(),
            timestamp=fake.past_date(),
            autor=u) db.session.add(p)
    db.session.commit()
```

Atribute ovih lažnih objekata proizvode nasumični generatori informacija koje obezbeđuje Faker paket, koji može generisati imena koja izgledaju stvarno, e-poštu, rečenice i mnoge druge atribute.

E-mail adrese i korisnička imena korisnika moraju biti jedinstvena, ali budući da ih Faker generiše na potpuno nasumičan način, postoji rizik od duplikata. U malo vjerovatnom slučaju da se generira duplikat, urezivanje sesije baze podataka će baciti izuzetak IntegrityError . Izuzetak se rješava vraćanjem sesije unatrag radi otkazivanja tog duplikata korisnika. Petlja će se izvoditi sve dok se ne generira traženi broj jedinstvenih korisnika.

Generisanje nasumičnih objava mora svakoj objavi dodijeliti slučajnog korisnika. Za ovo se koristi filter upita offset() . Ovaj filter odbacuje broj rezultata datih kao argument. Postavljanjem slučajnog pomaka i zatim pozivanjem first(), svaki put se dobija drugačiji slučajni korisnik.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11c da provjerite ovu verziju aplikacije. Da biste bili sigurni da imate instalirane sve zavisnosti, također pokrenite pip install -r requirements/dev.txt.

Nove funkcije olakšavaju kreiranje velikog broja lažnih korisnika i objava iz Python ljudske:

```
(venv) $ flask shell >>> iz
aplikacije uvoz lažni >>>
fake.users(100) >>>
fake.posts(100)
```

Ako sada pokrenete aplikaciju, vidjet ćete dugu listu nasumičnih postova na blogu na početnoj stranici, od mnogo različitih korisnika.

Rendering na stranicama

Primer 11-9 pokazuje promene na ruti početne stranice radi podrške paginacije.

Primjer 11-9. app/main/views.py: paginiranje liste blog postova

```
@main.route('/', methods=['GET', 'POST']) def
index():
    ...
    # page = request.args.get('page', 1, type=int) pagination
    = Post.query.order_by(Post.timestamp.desc()).paginate( page,
        per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
        postovi = pagination.items return render_template('index.html', form=form,
        posts=posts, pagination= pagination)
```

Broj stranice za prikaz dobija se iz upitnog niza zahteva, koji je dostupan kao `request.args`. Kada stranica nije data, koristi se zadana stranica od 1 (prva stranica). Argument `type=int` osigurava da ako se argument ne može pretvoriti u cijeli broj, vraća se zadana vrijednost.

Za učitavanje jedne stranice zapisa, konačni poziv `all()` metode objekta upita zamjenjuje se Flask-SQLAlchemy `paginate()`. Metoda `paginate()` uzima broj stranice kao prvi i jedini potrebni argument. Opcioni argument `per_page` se može dati da označi veličinu svake stranice, u broju stavki. Ako ovaj argument nije naveden, zadana vrijednost je 20 stavki po stranici. Drugi opcioni argument koji se zove `error_out` može se postaviti na `True` (podrazumevano) za izdavanje greške koda 404 kada se traži stranica izvan važećeg opsega. Ako je `error_out False`, stranice izvan važećeg raspona se vraćaju s praznom listom stavki. Da bi se veličine stranica mogle konfigurirati, vrijednost argumenta `per_page` se čita iz konfiguracijske varijable specifične za aplikaciju pod nazivom `FLASKY_POSTS_PER_PAGE` koja se dodaje u `cong.py`.

Sa ovim promjenama, lista blog postova na početnoj stranici će prikazati ograničen broj stavki. Da vidite drugu stranicu postova, dodajte string upita `?page=2` u URL u adresnoj traci pretraživača.

Dodavanje widgeta za paginaciju

Povratna vrijednost `paginate()` je objekt klase `Pagination`, klase koju definira Flask-SQLAlchemy. Ovaj objekt sadrži nekoliko svojstava koja su korisna za generiranje veza stranica u predlošku, pa se prosljeđuje predlošku kao argument. Sažetak atributa objekta paginacije prikazan je u [Tabeli 11-1](#).

Tabela 11-1. Flask-SQLAlchemy atributi objekta paginacije

Atribut	Opis
stavke	Zapisi na trenutnoj stranici
upit	Izvorni upit koji je paginiran
stranica	Broj trenutne stranice
prev_num	Broj prethodne stranice
next_num	Broj sljedeće stranice
has_next	Tačno ako postoji sljedeća stranica
has_prev	Tačno ako postoji prethodna stranica
stranice	Ukupan broj stranica za upit per_page
Broj stavki po stranici	
ukupno	Ukupan broj stavki koje je vratio upit

Objekt paginacije također ima neke metode, navedene u [Tabeli 11-2.](#)

Tabela 11-2. Metode objekata paginacije Flask-SQLAlchemy

Metoda	Opis
iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)	Iterator koji vraća niz brojeva stranica za prikaz u widgetu za paginaciju. Lista će imati stranice lijevo_ivice na lijevoj strani, lijevo_trenutne stranice lijevo od trenutne stranice, desne_trenutne stranice desno od trenutne stranice i desne_ivice stranice na desnoj strani. Na primjer, za stranicu 50 od 100 ovaj iterator konfiguriran sa zadanim vrijednostima će vratiti sljedeće stranice: 1, 2, Ništa, 48, 49, 50, 51, 52, 53, 54, 55, Ništa, 99, 100. A Ništa vrijednost u nizu označava prazninu u nizu stranica.
prev()	Objekt paginacije za prethodnu stranicu.
sljedeći()	Objekat paginacije za sljedeću stranicu.

Naoružani ovim moćnim objektom i Bootstrapovim CSS klasama paginacije, prilično je lako napraviti podnože paginacije u predlošku. Implementacija prikazana u [primjeru 11-10](#) je urađena kao višekratni Jinja2 makro.

Primjer 11-10. app/templates/_macros.html: makro šablona za paginaciju

```
{% macro pagination_widget(paginacija, krajnja tačka) %} <ul  
class="pagination">  
    <li{% if not paginacija.has_prev %} class="disabled"{% endif %}> <a href="{% if  
        paginacija.has_prev %}{% url_for(endpoint,  
            stranica = paginacija.page - 1, **kwargs) %}{% else %}{% endif %}"> &laquo; </a>  
    </li> {% za p u paginacija.iter_pages() %}
```

```

{%- if p %}

{%- if p == pagination.page %} <li
    class="active">
        <a href="{{ url_for(endpoint, page = p, **kwargs) }}>{{ p }}</a> </li> {%- else %}

<li> <a href="{{ url_for(endpoint, page = p, **kwargs) }}>{{ p }}</a> </li> {%- endif %}

{%- else %} <li class="disabled"><a href="#">&hellip;</a></li> {%- endif %} {%- endfor %}

<li{%- if not pagination.has_next %} class="disabled"{%- endif %}> <a href="{{% if
    pagination.has_next %}}{{ url_for(endpoint,

```

stranica = paginacija.stranica + 1, **kwargs) }}{%-else %}{%-endif %}"> & raquo ;

Makro kreira Bootstrap element paginacije, koji je stilizovana neuređena lista. Definira sljedeće linkove stranica unutar njega:

- Link "prethodna stranica". Ova veza dobija onemogućenu CSS klasu ako je trenutna stranica je prva stranica.
- Veze ka svim stranicama koje vraća iter_pages() objekta paginacije .
Ove stranice se prikazuju kao veze sa eksplisitim brojem stranice, datim kao argument url_for(). Stranica koja je trenutno prikazana je istaknuta pomoću aktivne CSS klase. Praznine u redoslijedu stranica prikazane su znakom elipse. • Link "sljedeća stranica". Ova veza će se pojaviti onemogućena ako je trenutna stranica posljednja stranica.

Jinja2 makroi uvijek primaju argumente ključne riječi bez potrebe za uključivanjem **kwargs u listu argumenata. Makro paginacije proslijedi sve argumente ključne riječi koje primi pozivu url_for() koji generiše veze za paginaciju. Ovaj pristup se može koristiti sa rutama kao što je stranica profila koje imaju dinamički dio.

Makro pagination_widget može se dodati ispod predloška _posts.html uključenog u index.html i user.html. **Primjer 11-11** pokazuje kako se koristi na početnoj stranici aplikacije.

Primjer 11-11. app/templates/index.html: podnožje paginacije za liste postova na blogu

```
{% proširuje "base.html" %} {%
  uvoz "bootstrap/wtf.html" kao wtf %} {% uvoz
  "_macros.html" kao makroe %}
...
{% include '_posts.html' %} <div
class="pagination">
{{ macros.pagination_widget(pagination, '.index') }} </div> {%
endif %}
```

Slika 11-2 pokazuje kako se veze paginacije pojavljuju na stranici.



Slika 11-2. Paginacija blog postova



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout
11d da provjerite ovu verziju aplikacije.
cija.

Objave obogaćenog teksta sa Markdown i Flask-PageDown

Postovi u običnom tekstu dovoljni su za kratke poruke i ažuriranje statusa, ali korisnici koji žele pisati duže članke smatraće da nedostatak formatiranja vrlo ograničava. U ovom odeljku, polje za tekst u koje se unose postovi biće nadograđeno da podrži smanjenje sintaksu i predstaviti pregled obogaćenog teksta posta.

Implementacija ove funkcije zahtjeva nekoliko novih paketa:

- PageDown, konverter Markdown-to-HTML na strani klijenta implementiran u Java-Skripta
- Flask-PageDown, omotač PageDown za Flask koji integriše PageDown sa Flask-WTF obrasci
- Markdown, konverter Markdown-to-HTML na strani servera implementiran u Python
- Bleach, HTML sanitizer implementiran u Python

Svi Python paketi se mogu instalirati sa pip-om:

```
(venv) $ pip install flask-pagedown markdown bleach
```

Korišćenje Flask- PageDown

Ekstenzija Flask-PageDown definiše klasu PageDownField koja ima isti interfejs kao TextAreaField iz WTForms. Prije nego što se ovo polje može koristiti, ekstenziju treba inicijalizirati kao što je prikazano u [primjeru 11-12](#).

Primjer 11-12. app/__init__.py: Flask-PageDown inicijalizacija

```
from flask_pagedown import PageDown #
pagedown = PageDown() #

...
def create_app(config_name):
    ...
    # pagedown.init_app(app)
    # ...
```

Da biste konvertovali kontrolu oblasti teksta na početnoj stranici u Markdown uređivač bogatog teksta, polje tela PostForma mora biti promjenjeno u PageDownField kao što je prikazano u [Primeru 11-13](#).

Primjer 11-13. app/main/forms.py: Obrasca za objavu s omogućenom Markdownom

```
iz flask_pagedown.fields import PageDownField

class PostForm(FlaskForm):
    body = PageDownField("Šta ti je na umu?", validators=[Obavezno()])
    submit = SubmitField('Pošalji')
```

Markdown pregled se generiše uz pomoć PageDown biblioteka, tako da se one moraju dodati u šablon. Flask-PageDown pojednostavljuje ovaj zadatak pružanjem

uzimanje šablonskog makroa koji uključuje potrebne datoteke sa CDN-a kao što je prikazano u [primjeru 11-14.](#)

Primjer 11-14. app/templates/index.html: deklaracija predloška Flask-PageDown

```
{% block skripti %}  
{{ super() }}  
{{ pagedown.include_pagedown() }} {%  
endblock %}
```



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11e da provjerite ovu verziju aplikacije. Da biste bili sigurni da imate instalirane sve ovisnosti, također pokrenite pip install -r requirements/dev.txt.

Sa ovim promjenama, Markdown formatirani tekst upisan u polje za tekst će se odmah prikazati kao HTML u području za pregled ispod. [Slika 11-3](#) prikazuje obrazac za prijavu bloga sa obogaćenim tekstrom.

Slika 11-3. Obrazac za blog sa obogaćenim tekstrom

Rukovanje obogaćenim tekstrom na

serveru Kada se obrazac pošalje, sa POST zahtevom se šalje samo sirovi Markdown tekst; HTML pregled koji je prikazan na stranici se odbacuje. Slanje generisanog HTML pregleda sa formom može se smatrati sigurnosnim rizikom, jer bi napadaču bilo prilično lako da konstruiše HTML sekvene koje se ne podudaraju sa Markdown izvorom i pošalje ih. Da bi se izbjegli bilo kakvi rizici, šalje se samo Markdown izvorni tekst, a kada se nađe na serveru, on se ponovo konvertuje u HTML koristeći Markdown, Python Markdown-to-HTML pretvarač. Rezultirajući HTML se čisti pomoću Bleach-a kako bi se osiguralo da se koristi samo kratka lista dozvoljenih HTML oznaka.

Pretvaranje postova na blogu Markdown u HTML može se obaviti u predlošku _posts.html, ali to je neefikasno jer će postovi morati da se konvertuju svaki put kada budu prikazani na stranici. Da bi se izbjeglo ovo ponavljanje, konverzija se može izvršiti jednom kada se blog post kreira, a zatim kešira u bazu podataka. HTML kod za prikazani blog post je keširan u novom polju dodanom modelu Post kojem šablon može direktno pristupiti. Originalni Markdown izvor se također čuva u bazi podataka u slučaju da objavu treba urediti. **Primjer 11-15** pokazuje promjene modela Post .

Primjer 11-15. app/models.py: Rukovanje tekstrom Markdown u modelu Post

```
from markdown import markdown
import bleach

class Post(db.Model):
    body_html =
        db.Column(db.Text) #
        ...
    @staticmethod
    def on_changed_body(cilj, vrijednost, stara vrijednost, pokretač):
        dozvoljeno_tags = ['a', 'abbr', 'akronim', 'b', 'navodnik', 'kod', 'em', 'i', 'li', 'ol', 'pre',
                           'jako', 'ul', 'h1', 'h2', 'h3', 'p']

        target.body_html = bleach.linkify(bleach.clean(markdown(value,
                                                               output_format='html'), tags=allowed_tags, strip=True))

    db.event.listen(Post.body, 'set', Post.on_changed_body)
```

Funkcija on_changed_body() je registrirana kao slušatelj SQLAlchemy-jevog "set" događaja za tijelo, što znači da će biti automatski pozvana kad god se polje tijela postavi na novu vrijednost. Funkcija rukovatelja prikazuje HTML verziju tijela i pohranjuje je u body_html, efektivno čineći konverziju Markdown teksta u HTML potpuno automatskom.

Stvarna konverzija se vrši u tri koraka. Prvo, funkcija markdown() vrši početnu konverziju u HTML. Rezultat se prosljeđuje clean(), zajedno sa listom

odobrene HTML oznake. Clean() funkcija uklanja sve oznake koje nisu na bijeloj listi. Konačna konverzija se vrši pomoću linkify(), još jedne funkcije koju pruža Bleach i koja pretvara sve URL-ove napisane u običnom tekstu u ispravne <a> veze. Ovaj poslednji korak je neophodan jer automatsko generisanje veze nije zvanično u Markdown specifikaciji, ali je veoma zgodna karakteristika. Na strani klijenta, PageDown podržava ovu funkciju kao opcionalnu ekstenziju, tako da linkify() odgovara toj funkciji na server.

Posljednja promjena je zamjena post.body sa post.body_html u predlošku kada je dostupan, kao što je prikazano u [primjeru 11-16](#).

Primjer 11-16. app/templates/_posts.html: koristite HTML verziju tijela postova u predlošku

```
...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | sigurno }} {%
    ostalo %} {{ post.body }} {% endif %} </
    div>
```

...

The | siguran sufiks prilikom prikazivanja HTML tijela je tu da kaže Jinja2 da ne izbjegava HTML elemente. Jinja2 podrazumevano izbegava sve varijable šablona kao bezbednosnu mjeru, ali HTML koji je generisao Markdown je generisao server, tako da je bezbedno da se renderuje direktno kao HTML.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11f da provjerite ovu verziju aplikacije. Ovo ažuriranje također sadrži migraciju baze podataka, pa ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod. Da biste bili sigurni da imate instalirane sve ovisnosti, također pokrenite pip install -r requirements/dev.txt.

Trajne veze do postova na blogu

Korisnici će možda htjeti podijeliti veze do određenih postova na blogu sa prijateljima na društvenim mrežama. U tu svrhu, svakom postu će biti dodijeljena stranica s jedinstvenim URL-om koji ga upućuje. Ruta i funkcija pregleda koje podržavaju trajne veze prikazane su u [primjeru 11-17](#).

Primjer 11-17. app/main/views.py: omogućavanje trajnih veza do objava

```
@main.route('/post/<int:id>') def
post(id): post = Post.query.get_or_404(id)
    return render_template('post.html',
    posts=[post])
```

URL-ovi koji će biti dodijeljeni objavama na blogu se konstruiraju sa jedinstvenim poljem id koji se dodjeljuje kada se objava ubaci u bazu podataka.



Za neke vrste aplikacija, izgradnja trajnih veza koje koriste čitljive URL-ove umjesto numeričkih ID-ova može biti poželjna. Alternativa numeričkim ID-ovima je da se svakoj objavi na blogu dodijeli puž, koji je jedinstveni niz koji se zasniva na naslovu ili prvih nekoliko riječi posta.

Imajte na umu da predložak post.html prima listu s jednim elementom koji je objava za renderiranje. Slanje liste je stvar pogodnosti, tako da se šablon _posts.html na koji upućuju index.html i user.html može koristiti i na ovoj stranici.

Stalne veze se dodaju na dno svake objave u generičkom predlošku _posts.html, kao što je prikazano u [primjeru 11-18](#).

Primjer 11-18. app/templates/_posts.html: dodavanje trajnih linkova na objave

```
<ul class="posts"> {%
    za objavu u objavama %} <li
        class="post">
            ...
            <div class="post-content">
                ...
                <div class="post-footer">
                    <a href="{{ url_for('.post', id=post.id) }}>
                        <span class="label label-default">Permalink </a> </div>
                </div> {%
            endfor %} </ul>
```

Novi predložak post.html koji prikazuje trajnu stranicu veze prikazan je u [primjeru 11-19](#). Uključuje primjer šablona.

Primjer 11-19. app/templates/post.html: trajni šablon veze

```
{% proširuje "base.html" %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %} {%
include '_posts.html' %} {%
endblock %}
```



Ako ste klonirali Git spremiše aplikacije na GitHub-u, možete pokrenuti git checkout 11g da provjerite ovu verziju aplikacije.

Urednik blogova

Posljednja karakteristika vezana za objave na blogu je uređivač postova koji omogućava korisnicima da uređuju svoje postove. Uređivač blog postova živi na samostalnoj stranici i takođe je zasnovan na Flask PageDown, tako da tekstualno područje u kojem se može uređivati Markdown tekst blog posta prati renderovani pregled. Šablon edit_post.html prikazan je u [primjeru 11-20](#).

Primjer 11-20. app/templates/edit_post.html: uredi predložak objave na blogu

```
{% proširuje "base.html" %} {%
import "bootstrap/wtf.html" kao wtf %}

{% block title %}Flasky - Uredi objavu{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Uredi objavu </h1> </
<div><div>
{{ wtf.quick_form(form) }} </
<div> {% endblock % }

{%
blok skripti %}
{{ super() }}
{{ pagedown.include_pagedown() }} {%
endblock %}
```

Ruta koja podržava uređivač postova na blogu prikazana je u [primjeru 11-21](#).

Primjer 11-21. app/main/views.py: uredi rutu posta na blogu

```
@main.route('/edit/<int:id>', methods=['GET', 'POST']) @login_required
def edit(id): post = Post.query.get_or_404(id) ako trenutni_user != post.author i \ ne current_user.can(Permission.ADMIN): abort(403)
    form = PostForm() ako form.validate_on_submit():

        post.body = form.body.data
        db.session.add(post)
        db.session.commit() flash('Objava je ažurirana.') return
            redirect(url_for('.post', id=post.id)) form.body.data =
        post.body return render_template('edit_post.html', form=form)
```

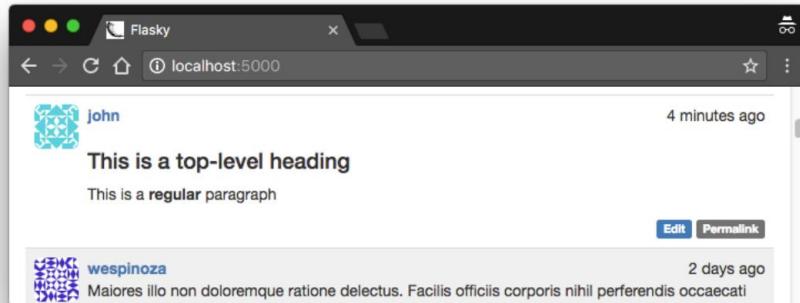
Ova funkcija pregleda je kodirana tako da dozvoljava samo autoru blog posta da ga uređuje, osim administratorima, kojima je dozvoljeno uređivati postove svih korisnika. Ako korisnik pokuša urediti objavu drugog korisnika, funkcija pregleda odgovara kodom 403. Klasa veb obrasca PostForm koja se koristi ovde je ista ona koja se koristi na početnoj stranici.

Da biste dovršili ovu funkciju, link ka uređivaču blog postova može se dodati ispod svakog blog posta, pored trajne veze, kao što je prikazano u [primjeru 11-22](#).

Primjer 11-22. app/templates/_posts.html: dodavanje veze za uređivanje blog posta

```
<ul class="posts"> {%
    za objavu u objavama %} <li
        class="post">
            ...
            <div class="post-content">
                ...
                <div class="post-footer">
                    ...
                    {% if current_user == post.author %} <a
                        href="{{ url_for('.edit', id=post.id) }}">
                            <span class="label label-primary">Uredi </a> {% elif
                        current_user.is_administrator() %} <a href="{{ url_for('.edit',
                        id=post.id) }}">
                            <span class="label label-danger">Uredi [Administrator] </a> {% endif %} </div> </div> </li> {% endfor %} </ul>
```

Ova promjena dodaje vezu "Uredi" na sve postove na blogu čiji je autor trenutni korisnik. Za administratore, link se dodaje na sve objave. Administratorska veza je drugačije stilizovana kao vizuelni znak da je ovo funkcija administracije. [Slika 11-4](#) prikazuje kako veze Uredi i Permalink izgledaju u web pretraživaču.



Slika 11-4. Uredite i stalne veze u postu na blogu



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 11h da provjerite ovu verziju aplikacije.

Followers

Društveno syesne web aplikacije omogućavaju korisnicima da se povežu s drugim korisnicima. Različite aplikacije nazivaju ove veze sljedbenicima, prijateljima, kontaktima, vezama ili prijateljima, ali funkcija je ista bez obzira na ime i u svim slučajevima uključuje praćenje usmjerenih veza između parova korisnika i korištenje ovih veza u upiti baze podataka.

U ovom poglavlju ćete naučiti kako da implementirate funkciju pratioca za Flasky. Korisnici će moći "pratiti" druge korisnike i odabrati filtriranje liste blog postova na početnoj stranici kako bi uključili samo one od korisnika koje prate.

Revisited Revisions Relationships

Kao što je objašnjeno u [poglavlju 5](#), baze podataka uspostavljaju veze između zapisa koristeći relacije. Odnos jedan-prema-više je najčešći tip odnosa, gdje je zapis povezan sa listom povezanih zapisa. Za implementaciju ove vrste odnosa, elementi na strani „mnogo“ imaju strani ključ koji ukazuje na povezani element na strani „jedan“. Primjer aplikacije u svom trenutnom stanju uključuje dva odnosa jedan prema više: jedan koji povezuje korisničke uloge sa listama korisnika i drugi koji povezuje korisnike s blog postovima čiji su autori.

Većina drugih tipova odnosa može biti izvedena iz tipa jedan-prema-više. Odnos „mnogo-na-jedan“ je odnos „jedan-na-više“ sa stanovišta „mnogo“ strane. Tip odnosa jedan-na-jedan je pojednostavljenje tipa jedan-prema-više, gdje je strana "mnogo" ograničena da ima najviše jedan element. Jedini tip odnosa koji se ne može implementirati kao jednostavna varijacija modela jedan-prema-više je mnogo-prema-više, koji ima liste elemenata na obje strane. Ovaj odnos je detaljno opisan u sljedećem odjeljku.

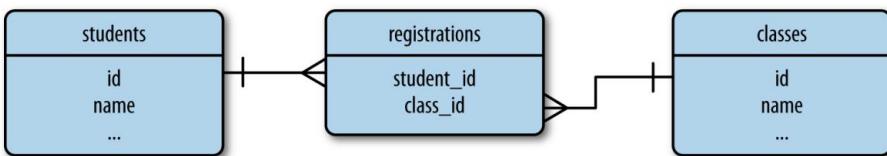
Relacije mnogo-prema-više Relacije

jedan-prema-više, mnogo-prema-jedan i jedan-na-jedan imaju barem jednu stranu s jednim entitetom, tako da se veze između povezanih zapisa implementiraju sa stranim ključevima koji upućuju na taj jedan element. Ali kako implementirati odnos u kojem su obje strane „mnogo“ strana?

Razmotrimo klasičan primjer odnosa više-prema-više: bazu podataka učenika i časova koje pohađaju. Jasno je da ne možete dodati strani ključ razreda u tabeli učenika, jer učenik pohađa mnogo časova — jedan strani ključ nije dovoljan.

Isto tako, ne možete dodati strani ključ učeniku u tabelu odeljenja, jer razred ima više od jednog učenika. Obje strane trebaju listu stranih ključeva.

Rješenje je dodavanje treće tablice u bazu podataka, koja se zove tabela asocijacija. Sada se odnos „više-prema-više“ može razložiti u dva odnosa „jedan-prema-više“ iz svake od dvije originalne tablice u asocijsku tablicu. [Slika 12-1](#) pokazuje kako je predstavljen odnos više-prema-više između učenika i razreda.



Slika 12-1. Primjer odnosa "mnogo prema mnogo".

Tablica asocijacija u ovom primjeru se zove registracije. Svaki red u ovoj tabeli predstavlja individualnu registraciju učenika u razredu.

Ispitivanje odnosa više-prema-više je proces u dva koraka. Da biste dobili listu časova koje student pohađa, polazite od odnosa jedan-prema-više između učenika i prijava i dobijate listu upisa za željenog studenta. Tada se odnos jedan-prema-više između razreda i registracija prelazi u smjeru više-prema-jedan kako bi se dobile sve klase povezane s registracijama dohvaćenim za učenika. Isto tako, da biste pronašli sve učenike u razredu, krenete od razreda i dobijete listu registracija, a zatim povežete učenike sa tim registracijama.

Prolazak kroz dva odnosa da bi se dobili rezultati upita zvuči teško, ali za jednostavnu relaciju kao što je ona u prethodnom primjeru, SQLAlchemy obavlja većinu posla. Slijedi kod koji predstavlja odnos više-prema-više na [slici 12-1](#):

```

registracije = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

razred učenik(db.Model):
    id = db.Column(db.Integer, primary_key=True) name
    = db.Column(db.String) classes = db.relationship('Class',
        secondary=registracije, backref=db.backref('students',
            lazy='dynamic'),
            läjen='dynamic')
)

```

klasa Class(db.Model):

```

id = db.Column(db.Integer, primary_key=True) ime =
db.Column(db.String)

```

Odnos je definiran istom konstrukcijom db.relationship() koja se koristi za odnose jedan-prema-više, ali u slučaju relacije više-prema-više dodatni sekundarni argument mora biti postavljen na tablicu asocijacija. Odnos se može definirati u bilo kojoj od dvije klase, s argumentom backref koji vodi računa o razotkrivanju odnosa i s druge strane. Tablica asocijacija je definirana kao jednostavna tablica, a ne kao model, budući da SQLAlchemy upravlja ovom tablicom interno.

Odnos klasa koristi semantiku liste, što čini rad sa odnosom mnogo-prema mnogo konfigurisanim na ovaj način izuzetno lakis. S obzirom na učenike s i razred c, kod koji registruje učenika za razred je:

```

>>> s.classes.append(c) >>>
db.session.add(s)

```

Upiti koji navode razrede za koje je učenik s prijavljen i spisak učenika prijavljenih za razred c su također vrlo jednostavnii:

```

>>> s.classes.all() >>>
c.students.all()

```

Odnos učenika dostupan u modelu klase je onaj definiran u argumentu db.backref(). Imajte na umu da je u ovom odnosu argument backref proširen tako da ima atribut lazy='dynamic', tako da obje strane vraćaju upit koji može prihvati dodatne filtere.

Ako učenik kasnije odluči napustiti razred c, možete ažurirati bazu podataka na sljedeći način:

```

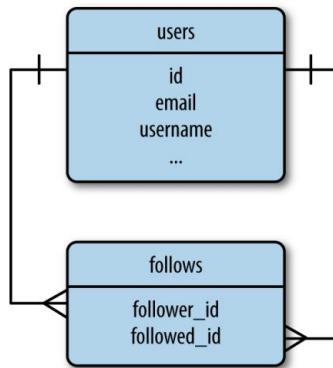
>>> s.classes.remove(c)

```

Samoreferentni odnosi Odnos više-

prema-više može se koristiti za modeliranje korisnika koji slijede druge korisnike, ali postoji problem. Na primjeru učenika i odjeljenja, postojala su dva vrlo jasno definirana entiteta povezana tabelom asocijacije. Međutim, za predstavljanje korisnika koji slijede druge korisnike, to su samo korisnici — ne postoji drugi entitet.

Za odnos u kojem obje strane pripadaju istom stolu se kaže da je samoreferencijalan. U ovom slučaju entiteti na lijevoj strani odnosa su korisnici, koji se mogu nazvati "sljedbenicima". Entiteti na desnoj strani su također korisnici, ali to su „praćeni“ korisnici. Konceptualno, samoreferentni odnosi se ne razlikuju od redovnih, ali o njima je teže razmišljati. **Slika 12-2** prikazuje dijagram baze podataka za samoreferentni odnos koji predstavlja korisnike koji slijede druge korisnika.



Slika 12-2. Sljedbenici, odnos mnogo-prema-više

Tablica asocijacija u ovom slučaju se zove kako slijedi. Svaki red u ovoj tabeli predstavlja korisnika koji prati drugog korisnika. Odnos jedan-prema-više prikazan na lijevoj strani povezuje korisnike sa listom "prati" redova u kojima su oni sljedbenici. Odnos jedan-prema-više prikazan na desnoj strani povezuje korisnike sa listom "prati" redova u kojima su oni praćeni korisnik.

Napredni odnosi „mnogo-prema-više“ Sa

samoreferencijalnom relacijom „mnogo-prema-više“ konfiguriranom kao što je prikazano u prethodnom primjeru, baza podataka može predstavljati sljedbenike – ali postoji jedno ograničenje. Uobičajena potreba kada se radi sa odnosima mnogo-prema-više je pohranjivanje dodatnih podataka koji se odnose na vezu između dva entiteta. Za odnos sljedbenika, može biti korisno pohraniti datum kada je korisnik počeo pratiti drugog korisnika, jer će to omogućiti da se liste sljedbenika predstave hronološkim redom. Jedino mjesto ove informacije je u tablici asocijacija.

može biti pohranjena u tablici asocijacija, ali u implementaciji sličnoj onoj za učenike i razrede prikazanu ranije, tablica asocijacija je interna tablica kojom u potpunosti upravlja SQLAlchemy.

Da biste mogli raditi s prilagođenim podacima u odnosu, tabela asocijacija mora biti unapređena u odgovarajući model kojem aplikacija može pristupiti. [Primjer 12-1](#) prikazuje novu tablicu asocijacija, predstavljenu modelom Follow .

Primjer 12-1. app/models.py: sljedeća tablica asocijacija kao model

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    vremenska_oznaka = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy ne može transparentno koristiti tablicu asocijacija jer to neće dati aplikaciji pristup prilagođenim poljima u njoj. Umjesto toga, odnos „mnogo-prema-više“ mora se razložiti na dva osnovna odnosa jedan-prema-više za lijevu i desnu stranu, a oni moraju biti definirani kao standardni odnosi. To je prikazano u [primjeru 12-2](#).

Primjer 12-2. app/models.py: odnos više-prema-više implementiran kao dva odnosa jedan-prema-više

```
class User(UserMixin, db.Model):
    ...
    # followed = db.relationship('Follow',
    #                            external_keys=[Follow.follower_id],
    #                            backref=db.backref('follower', lazy='joined'), lazy='dynamic',
    #                            cascade='all, delete-orphan')
    # followers =
    #     db.relationship('Follow', external_keys=[Follow.followed_id],
    #                    backref=db.backref('followed', lazy='joined'), lazy='dynamic', cascade='all, delete- siroče')
```

Ovdje se odnosi praćenja i sljedbenika definiraju kao pojedinačni odnosi jedan prema više. Imajte na umu da je neophodno eliminisati bilo kakvu dvostrukturu između stranih ključeva tako što ćete u svakoj relaciji navesti koji strani ključ koristiti preko neobaveznog argumenta external_keys . Argumenti db.backref() u ovim relacijama ne važe jedni na druge; zadnje reference se primjenjuju na model Follow .

Lijeni argument za povratne reference je specificiran kao spojen . Ovaj lijeni način uzrokuje da se povezani objekt odmah učita iz upita za spajanje. Na primjer, ako korisnik prati 100 drugih korisnika, pozivanje user.followed.all() će vratiti listu od 100 instanci praćenja , gdje svaka ima svojstva sljedbenika i praćene povratne reference postavljene na odgovarajuće korisnike. Lazy ='joined' način omogućava da se sve ovo dogodi iz jednog upita baze podataka. Ako je lazy postavljena na zadanu vrijednost select, onda se pratilac i praćeni korisnici učitavaju lijeno kada im se prvi put pristupi i svaki atribut će zahtijevati pojedinačni upit, što znači da bi za dobijanje kompletne liste praćenih korisnika bilo potrebno 100 dodatnih upita baze podataka .

Lijeni argument na strani korisnika oba odnosa ima različite potrebe. Oni su na strani "jedan" i vraćaju stranu "mnogo"; ovdje se koristi način dinamike , tako da atributi odnosa vraćaju objekte upita umjesto da direktno vraćaju stavke. Ovo omogućava da se upitu dodaju dodatni filteri prije nego što se izvrši.

Argument kaskade konfiguriše kako se akcije izvedene na roditeljskom objektu šire na povezane objekte. Primjer kaskadne opcije je pravilo koje kaže da kada se objekt doda sesiji baze podataka, svi objekti povezani s njim kroz relacije također treba automatski biti dodati sesiji. Zadane kaskadne opcije su prikladne za većinu situacija, ali postoji jedan slučaj u kojem zadane kaskadne opcije ne funkcionišu dobro za ovaj odnos mnogo-prema-više. Zadano kaskadno ponašanje kada se objekt briše je postavljanje stranog ključa u svim povezanim objektima koji se povezuju na njega na nultu vrijednost. Ali za tabelu asocijacija, ispravno ponašanje je brisanje unosa koji upućuju na zapis koji je obrisan, jer to efektivno uništava vezu. To je ono što radi kaskadna opcija brisanja siročeta .



Vrijednost koja se daje kaskadi je lista kaskadnih opcija odvojena zarezima. Ovo je pomalo zbunjujuće, ali opcija pod nazivom all predstavlja sve kaskadne opcije osim delete-orphan. Koristeći vrijednost all, delete-orphan ostavlja uključene zadane kaskadne opcije i dodaje ponašanje brisanja za siročad.

Aplikacija sada treba da radi sa dve relacije „jedan-prema-više“ da bi implementirala funkcionalnost „više-prema-više“. Pošto su to operacije koje će se morati često ponavljati, dobra je ideja kreirati pomoćne metode u modelu korisnika za sve moguće operacije. Četiri nove metode koje kontroliraju ovaj odnos prikazane su u **primjeru 12-3.**

Primjer 12-3. app/models.py: pomoćne metode sljedbenika

```

klasa Korisnik(db.Model):
    # ...
    def follow(self, user): ako
        nije self.is_following(user): f =
            Follow(follower=self, followed =user)
        db.session.add(f)

    def unfollow(self, user): f =
        self.followed.filter_by(followed_id=user.id).first() if f: db.session.delete(f)

    def is_following(self, user):
        ako je user.id None:
            return False return
        self.followed.filter_by( followed_id
            =user.id).first() nije Ništa

    def is_followed_by(self, user): ako je
        user.id None: return False return

        self.followers.filter_by( follower_id=user.id).first()
        nije Ništa

```

Metoda follow() ručno ubacuje instancu Prati u tablicu asocijacija koja povezuje sljedbenika sa praćenim korisnikom, dajući aplikaciji priliku da postavi prilagođeno polje. Dva korisnika koja se povezuju ručno se dodeljuju novoj instanci Follow u njenom konstruktoru, a zatim se objekat dodaje u sesiju baze podataka kao i obično. Imajte na umu da nema potrebe za ručno postavljanjem polja vremenske oznake jer je definirano sa zadanom vrijednošću koja postavlja trenutni datum i vrijeme. The

unfollow() metoda koristi odnos praćenja da locira instancu Prati koja povezuje korisnika sa praćenim korisnikom kojeg treba isključiti. Da bi se uništila veza između dva korisnika, objekt Follow se jednostavno briše. Metode is_following() i is_followed_by() pretražuju relacije jedan prema više na lijevoj i desnoj strani za datog korisnika i vraćaju True ako je korisnik pronađen. I jedan i drugi osiguravaju da je datom korisniku dodijeljen id prije izdavanja upita, kako bi se izbjegle greške ako je naveden korisnik koji je kreiran, ali još nije upisan u bazu podataka.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 12a da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

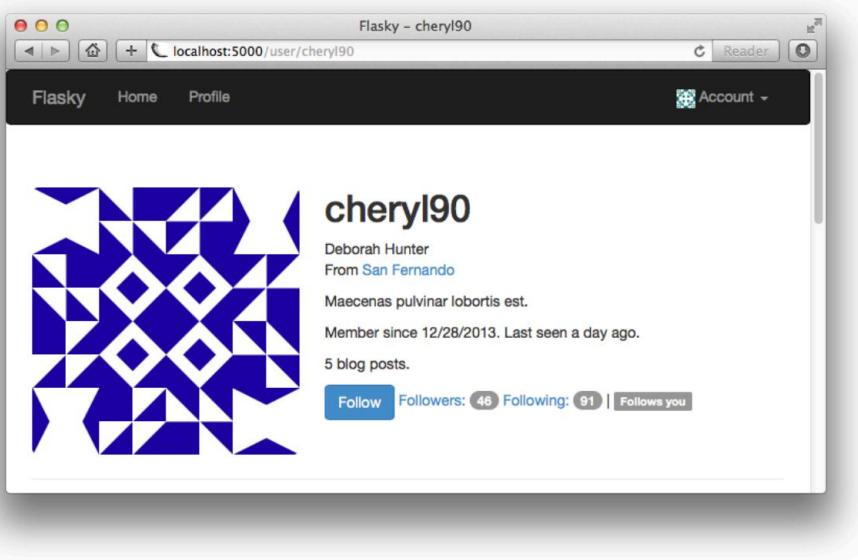
Dio baze podataka ove funkcije je sada završen. Možete pronaći jedinični test koji vježba novi odnos baze podataka u spremištu izvornog koda na GitHubu.

Pratioci na stranici Prole

Stranica profila korisnika treba da ima dugme „Prati“ ako korisnik koji je gleda nije pratilac ili dugme „Prekini praćenje“ ako je korisnik pratilac. Takođe je lep dodatak za prikazivanje broja pratilaca i praćenih, prikazivanje liste pratilaca i korisnika koji prate i prikazivanje znaka „Prati vas“ kada je to prikladno. Promjene predloška korisničkog profila prikazane su u **primjeru 12-4.** **Slika 12-3** prikazuje kako dodaci izgledaju na stranici profila.

Primjer 12-4. app/templates/user.html: poboljšanja sljedbenika u zaglavlju korisničkog proleta

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %} <a href="{{ url_for('.follow', korisničko_ime=user.username) }}" class="btn btn-primary">Prati</a> { % ostalo %} <a href="{{ url_for('.unfollow', korisničko_ime=user.username) }}>
        Prestani pratiti</a>
    {% endif %}
    {% endif %}
<a href="{{ url_for('.followers', korisničko_ime=user.username) }}>
    Sljedbenici: <span class="badge">{{ user.followers.count() }} </span> <a href="{{ url_for('.followed_by', korisničko_ime=user.username) }}>
        Slijedi: <span class="badge">{{ user.followed.count() }} </span> { % if current_user.is_authenticated and user != current_user and
            user.is_following(trenutni_user) %}
        | <span class="label label-default">Prati vas { % endif %}
```



Slika 12-3. Pratioci na prole stranici

Postoje četiri nove krajne tačke definirane u ovim promjenama predloška. Ruta /follow/<user-name> se poziva kada korisnik klikne na dugme „Follow“ na stranici profila drugog korisnika. Implementacija je prikazana u **primjeru 12-5.**

Primjer 12-5. app/main/views.py: pratite rutu i funkciju pregleda

```
@main.route('/follow/<korisničko ime>')
@login_required
@permission_required(Permission.FOLLOW) def
follow(username): user =
    User.query.filter_by(username=username).first() ako je korisnik
    Ništa: flash('Nevažeći korisnik.') return redirect(url_for('.index'))

    if current_user.is_following(user): flash(
        Već pratite ovog korisnika.) return redirect(url_for('.user',
        korisničko ime=korisničko ime))
    current_user.follow(user)
    db.session.commit() flash(
        Sada pratite %s.' % korisničko ime) return
    redirect(url_for('.user', username=username))
```

Ova funkcija pogleda učitava traženog korisnika, provjerava da li je važeći i da ga već ne prati prijavljeni korisnik, a zatim poziva pomoćnu funkciju follow() u

model korisnika za uspostavljanje veze. Ruta /unfollow/<username> implementirana je na sličan način.

Ruta /followers/<username> se poziva kada korisnik klikne na broj pratilaca drugog korisnika na stranici profila. Implementacija je prikazana u [primjeru 12-6](#).

Primjer 12-6. app/main/views.py: ruta pratilaca i funkcija pregleda

```
@main.route('/followers/<korisničko ime>')
def followers(username): user =
    User.query.filter_by(username=username).first() ako je korisnik
    Ništa: flash('Invalid user.') return . redirect(url_for('.index')) page =
        request.args.get('page', 1, type=int) pagination =
            user.followers.paginate(
                stranica, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
                error_out=False) slijedi = [ {'user': item.follower, 'timestamp': item.timestamp} za stavku u pagination.items ] return render_template('followers.html', user=user,
                    title="Sljedbenici",
                    endpoint='.followers', pagination=paginacija,
                    follows=follows )
```

Ova funkcija učitava i potvrđuje traženog korisnika, zatim paginira njegov odnos sljedbenika koristeći iste tehnike naučene u [poglavlju 11](#). Budući da upit za pratioca vraća instance Prati , lista se konvertuje u drugu listu koja ima polja korisnika i vremenske oznake u svakom unosu tako da renderovanje je jednostavnije.

Predložak koji prikazuje listu pratilaca može se napisati generički tako da se može koristiti za liste pratilaca i praćenih korisnika. Predložak prima korisnika, naslov stranice, krajnju tačku za korištenje u vezama za paginaciju, objekt paginacije i listu rezultata.

Krajnja tačka followed_by je skoro identična. Jedina razlika je u tome što se lista korisnika dobija iz odnosa user.followed . Argumenti predloška se također prilagođavaju u skladu s tim.

Šablon followers.html implementiran je sa tabelom od dvije kolone koja prikazuje korisnička imena i njihove avatare na lijevoj strani i Flask-Moment vremenske oznake na desnoj strani. Možete konsultovati spremište izvornog koda na GitHubu kako biste detaljno proučili implementaciju.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 12b da provjerite ovu verziju aplikacije.

Upitivanje praćenih postova pomoću pridruživanja bazi podataka

Početna stranica aplikacije trenutno prikazuje sve postove u bazi podataka u silaznom hronološkom redu. S obzirom da je funkcija pratilaca sada završena, bilo bi lijepo dati korisnicima mogućnost da gledaju postove na blogu samo onih korisnika koje prate.

Očigledan način da učitate sve postove čiji su autori praćeni korisnici je da prvo dobijete listu tih korisnika, a zatim dobijete postove od svakog i sortirate ih u jednu listu. Naravno, taj pristup nije dobro razmjeran; napor potreban za dobijanje ove kombinovane liste će rasti kako baza podataka bude rasla, a operacije kao što je paginacija ne mogu da se urade efikasno. Ovaj problem je uobičajeno poznat kao "N+1 problem", jer rad s bazom podataka na ovaj način zahtijeva izdavanje N+1 upita baze podataka, pri čemu je N broj rezultata koje je vratio prvi upit. Ključ za dobijanje postova na blogu sa dobrim performansama bez obzira na veličinu baze podataka je da sve to uradite jednim upitom.

Operacija baze podataka koja to može učiniti zove se spajanje. Operacija spajanja uzima dvije ili više tablica i pronalazi sve kombinacije redova koje zadovoljavaju dati uvjet. Rezultirajući kombinovani redovi se ubacuju u privremenu tabelu koja je rezultat spajanja. Najbolji način da objasnite kako spojevi funkcioniraju je kroz primjer.

Tabela 12-1 prikazuje primjer tablice korisnika sa tri korisnika.

Tabela 12-1. tabela korisnika

id korisničko ime
1 John
2 susan
3 David

Tabela 12-2 prikazuje odgovarajuću tabelu postova , sa nekim blog postovima.

Tabela 12-2. postova tabela

id autor_id tijelo
1 2 Blog post od Susan
2 1 Objava na blogu johna
3 3 Davidova objava na blogu
4 1 Drugi blog post od johna

Konačno, **tabela 12-3** pokazuje ko koga prati. U ovoj tabeli možete vidjeti da John prati Davida, Susan prati Johna i Davida, a David ne prati nikoga.

Tabela 12-3. prati tabelu

follower_id	followed_id
1	3
2	1
2	3

Da biste dobili listu postova korisnika iza kojih slijedi korisnik susan, tabele postova i praćenja moraju se kombinirati. Prvo se sljedeća tablica filtrira kako bi se zadržali samo redovi koji imaju susan kao sljedbenika, a to su u ovom primjeru posljednja dva reda. Zatim se kreira privremena tabela pridruživanja iz svih mogućih kombinacija redova iz postova i filtrira tabele praćenja u kojima je autor_id objave isti kao followed_id sljedećeg, efektivno birajući sve postove koji se pojavljuju na listi korisnici susan prati. **Tabela 12-4** prikazuje rezultat operacije spajanja. Kolone koje su korištene za izvođenje spajanja označene su znakom

* u ovoj tabeli.

Tabela 12-4. Spojeni sto

id	autor_id*	tijelo	follower_id	followed_id*
2	1	Objava na blogu johna	2	1
3	3	Davidova objava na blogu	2	3
4	1	Drugi blog post od Ivana	2	1

Ova tabela sadrži upravo listu blog postova čiji su autori korisnici koje Susan prati. Flask-SQLAlchemy upit koji izvodi operaciju spajanja kako je opisano prilično je složen:

```
return db.session.query(Post).select_from(Follow).\n    filter_by(follower_id=self.id)\.\join(Objava, Follow.followed_id\n    == Post.author_id)
```

Svi upiti koje ste do sada vidjeli počinju od atributa upita modela koji je upitan. Taj format ne radi dobro za ovaj upit, jer upit treba da vrati redove postova , ali prva operacija koju treba uraditi je da se primeni filter na sledeću tabelu. Dakle, umjesto toga se koristi osnovniji oblik upita. Da biste u potpunosti razumjeli ovaj upit, svaki dio treba pogledati zasebno:

- db.session.query(Post) specificira da će ovo biti upit koji vraća Objavite objekte.
- select_from(Follow) kaže da upit počinje modelom Follow . • filter_by(follower_id=self.id) vrši filtriranje sljedeće tabele prema korisnika pratnoga.

- join(Post, Follow.followed_id == Post.author_id) spaja rezultate filter_by() sa objektima Post .

Upit se može pojednostaviti zamjenom redoslijeda filtera i spajanja:

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\n    .filter(Followfollower_id == self.id)
```

Prvo izdavanje operacije spajanja znači da se upit može pokrenuti iz Post.query, tako da su sada jedina dva filtera koja treba primijeniti su join() i filter(). Može se činiti da bi prvo spajanje, a zatim filtriranje bilo više posla, ali u stvarnosti su ova dva upita ekvivalentna. SQLAlchemy prvo prikuplja sve filtere, a zatim generira upit na najefikasniji način. Nativne SQL instrukcije za ova dva upita su skoro identične, nešto što možete potvrditi ispisom objekta upita konvertovanog u string (tj. print(str(query))). Konačna verzija ovog upita dodaje se modelu Post , kao što je prikazano u [primjeru 12-7.](#)

[Primjer 12-7. app/models.py: dobijanje praćenih objava](#)

```
klasa Korisnik(db.Model):\n    # ...\n\n    @property\n    def followed_posts (self):\n        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\n            .filter(Followfollower_id == self.id)
```

Imajte na umu da je metoda followed_posts() definirana kao svojstvo tako da joj nije potreban (). Na taj način svi odnosi imaju konzistentnu sintaksu.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 12c da provjerite ovu verziju aplikacije.

Spojeve je izuzetno teško obaviti oko glave; možda ćete trebati eksperimentirati s primjerom koda u Ijusci prije nego što se sve utopi.

Prikaz praćenih postova na početnoj stranici

Početna stranica sada može dati korisnicima izbor da pogledaju sve postove na blogu ili samo one koje prate korisnici. [Primjer 12-8](#) pokazuje kako se ovaj izbor implementira.

Primjer 12-8. app/main/views.py: prikazuje sve ili praćene postove

```
@main.route('/', methods = ['GET', 'POST']) def index():

    ...
    # show_followed = Netačno
    ako je trenutni_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', '')) if show_followed:
            query = current_user.followed_posts ostalo:

            query = Post.query
            paginacija = query.order_by(Post.timestamp.desc()).paginate(
                stranica, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
                error_out=False) postovi = pagination.items return render_template('index.html',
            form=form, posts=posts, show_followed=show_followed, pagination =paginacija)
```

Izbor prikazivanja svih ili praćenih postova pohranjuje se u kolačić koji se zove show_followed koji, kada je postavljen na neprazan niz, označava da se trebaju prikazati samo praćeni postovi. Kolačići se pohranjuju u objektu zahtjeva kao rečnik request.cookies . Vrijednost niza kolačića se konvertuje u Boolean, a na osnovu njene vrijednosti lokalna varijabla upita postavlja se na upit koji dobiva potpunu ili filtriranu listu blog postova. Za prikaz svih postova koristi se upit najvišeg nivoa Post.query , a nedavno dodano svojstvo User.followed_posts se koristi kada lista treba biti ograničena na praćene korisnike. Upit pohranjen u lokalnoj varijabli upita se zatim paginira i rezultati se šalju u predložak kao i prije.

Kolačić show_followed postavljen je na dvije nove rute, prikazane u [primjeru 12-9](#).

Primjer 12-9. app/main/views.py: odabir svih ili praćenih objava

```
@main.route('/all')
@login_required def
show_all(): resp =
    make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60 *60) # 30 dana povrata odn

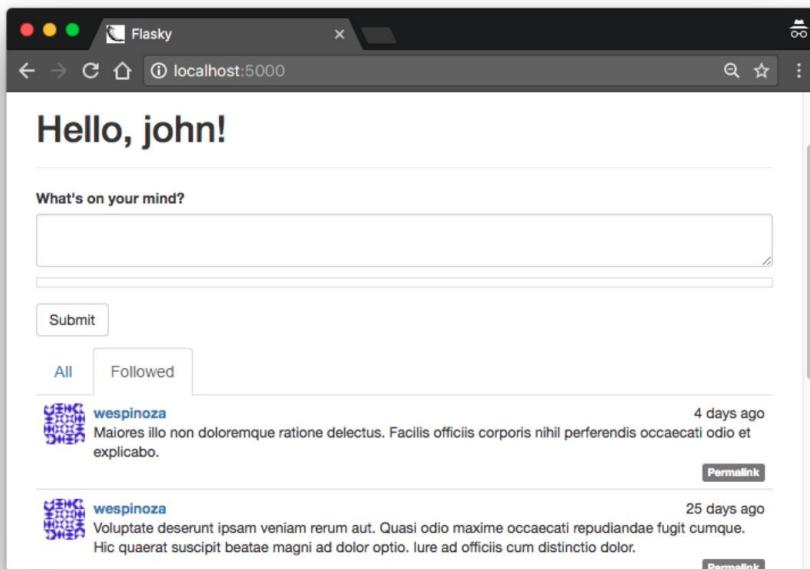
@main.route('/followed')
@login_required def
show_followed(): resp =
    make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24* 60*60) # 30 dana povrat odn
```

Veze ka ovim rutama se dodaju u šablon početne stranice. Kada se pozovu, kolačić show_followed se postavlja na odgovarajuću vrijednost i izdaje se preusmjeravanje natrag na početnu stranicu.

Kolačići se mogu postaviti samo na objekt odgovora, tako da ove rute moraju kreirati objekt odgovora preko make_response() umjesto da to dopuste Flasku.

Funkcija set_cookie() uzima ime kolačića i vrijednost kao prva dva argumenta. Opcijski argument max_age postavlja broj sekundi do isteka kolačića. Ako se ovaj argument ne uključi, kolačić će isteći kada se zatvori prozor pretraživača. U tom slučaju je postavljena maksimalna starost od 30 dana tako da se postavka pamti čak i ako se korisnik ne vrati u aplikaciju nekoliko dana.

Promjene u šablonu dodaju dvije navigacijske kartice na vrhu stranice koje pozivaju /all ili /followed rute za postavljanje ispravnih postavki u sesiji. Možete detaljno pregledati promjene predloška u spremištu izvornog koda na GitHubu. [Slika 12-4](#) prikazuje kako početna stranica izgleda sa ovim promjenama.



Slika 12-4. Pratili postove na početnoj stranici



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 12d da provjerite ovu verziju aplikacije.

Ako isprobate aplikaciju u ovom trenutku i prebacite se na sljedeću listu objava, primijetit ćete da se vaši postovi ne pojavljuju na listi. Ovo je naravno tačno, jer korisnici nisu sami sebi sljedbenici.

Iako upiti funkcioniraju kako je dizajnirano, većina korisnika će očekivati da vidi svoje vlastite postove kada gledaju objave svojih prijatelja. Najlakši način za rješavanje ovog problema je registracija svih korisnika kao vlastitih sljedbenika u trenutku kada su kreirani.

Ovaj trik je prikazan u [primjeru 12-10](#).

Primjer 12-10. app/models.py: pravljenje korisnika vlastitim sljedbenicima kada se kreiraju

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

Nažalost, vjerovatno imate nekoliko korisnika u bazi podataka koji su već kreirani i ne prate sami sebe. Ako je baza podataka mala i laka za regeneraciju, onda se može izbrisati i ponovo kreirati, ali ako to nije opcija, onda je dodavanje funkcije ažuriranja koja popravlja postojeće korisnike pravo rješenje. To je prikazano u [primjeru 12-11](#).

Primjer 12-11. app/models.py: pretvaranje korisnika u vlastite sljedbenike

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        korisnika u User.query.all():
            ako nije user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
        # ...
```

Sada se baza podataka može ažurirati pokretanjem prethodne funkcije primjera iz ljske:

```
(venv) $ flask shell >>>
User.add_self_follows()
```

Kreiranje funkcija koje uvode ažuriranja u bazu podataka uobičajena je tehnika koja se koristi za ažuriranje aplikacija koje su raspoređene, jer je pokretanje skriptiranog ažuriranja manje podložno greškama od ručnog ažuriranja baza podataka. U poglavljiju 17 vidjet ćete kako se ova funkcija i druge slične njoj mogu ugraditi u skriptu za implementaciju.

Omogućavanje da svi korisnici sami prate aplikaciju čini aplikaciju upotrebljivijom, ali ova promjena donosi nekoliko komplikacija. Broj pratilaca i praćenih korisnika prikazan na stranici korisničkog profila sada je povećan za jedan zbog linkova za samopratenje. Brojeve je potrebno smanjiti za jedan da bi bili tačni, što je lako uraditi direktno u šablonu renderiranjem `{{ user.followers.count() - 1 }}` i `{{ user.followed.count() - 1 }}` . Liste sljedbenika korisnika također se moraju prilagoditi tako da ne prikazuju istog korisnika, što je još jedan jednostavan zadatak koji treba obaviti u predlošku sa uslovom. Konačno, na sve jedinice testove koji provjeravaju broj sljedbenika također utiču veze samosljedbenika i moraju se prilagoditi kako bi se uračunali samosljedbenici.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 12e da provjerite ovu verziju aplikacije.

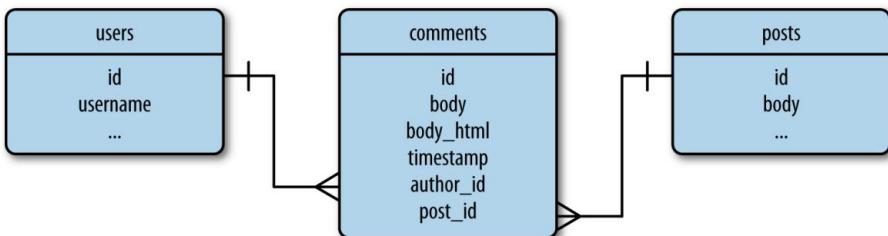
U sljedećem poglavlju bit će implementiran podsistem korisničkih komentara – još jedna vrlo važna karakteristika društveno svjesnih aplikacija.

Komentari korisnika

Omogućavanje korisnicima interakcije ključno je za uspjeh platforme za društveno blogovanje. U ovom poglavlju ćete naučiti kako implementirati komentare korisnika. Predstavljene tehnike su dovoljno generičke da se mogu direktno primijeniti na veliki broj društveno omogućenih aplikacija.

Reprezentacija komentara u bazi podataka

Komentari se ne razlikuju mnogo od blogova. Oba imaju tijelo, autora i vremensku oznaku, ali ovoj konkretnoj implementaciji oba su napisana Markdown sintaksom. [Slika 13-1](#) prikazuje dijagram tabele komentara i njenih odnosa sa drugim tabelama u bazi podataka.



Slika 13-1. Reprezentacija komentara na blogu u bazi podataka

Komentari se primjenjuju na određene postove na blogu, tako da je definiran odnos jedan-prema-više iz tabele postova . Ovaj odnos se može koristiti za dobijanje liste komentara povezanih sa određenim postom na blogu.

Tabela komentara je također u odnosu jedan-prema-više s tablicom korisnika . Ovaj odnos daje pristup svim komentarima koje je napravio korisnik, i posredno kako

mnogo komentara koje je korisnik napisao, informacija koja može biti zanimljiva za prikaz na stranicama korisničkog profila. Definicija modela komentara prikazana je u [primjeru 13-1](#).

Primjer 13-1. app/models.py: model komentara

```
class Komentar(db.Model):
    __tablename__ = 'komentari'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    vremenska_oznaka = db.Column(db.DateTime,
        index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(cilj, vrijednost, stara_vrijednost, pokretač):
        dozvoljeno_tags = ['a', 'abbr', 'akronim', 'b', 'kod', 'em', 'i', 'jaki']
        target.body_html = bleach.linkify(bleach.clean(markdown(value,
            output_format='html'), tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

Atributi modela komentara su skoro isti kao i atributi Post. Jedan dodatak je disabled polje,

Boolean koji će koristiti moderatori da potisnu komentare koji su neprikladni ili uvredljivi. Poput postova na blogu, komentari definiraju događaj koji se pokreće svaki put kada se promijeni polje tijela, automatizirajući prikazivanje Markdown teksta u HTML. Proces je identičan onome što je urađeno za postove na blogu u [poglavlju 11](#), ali pošto su komentari obično kratki, lista HTML označke koje su dozvoljene u konverziji iz Markdown-a je restriktivnija, označke vezane za paragraf su uklonjene i preostale su samo označke za formatiranje znakova.

Da bi se dovršile promjene baze podataka, modeli User i Post moraju definirati odnose jedan prema mnogo s tablicom komentara, kao što je prikazano u [Primjeru 13-2](#).

Primjer 13-2. app/models.py: odnosi jedan-prema-više korisnika i postova prema komentari

```
klasa Korisnik(db.Model):
    # ...
    comments = db.relationship('Komentar', backref='author', lazy='dynamic')

klasa Post(db.Model): #
    ...
    comments = db.relationship('Komentar', backref='post', lazy='dynamic')
```

Podnošenje i prikaz komentara

U ovoj aplikaciji, komentari se prikazuju na pojedinačnim stranicama blog postova koje su dodane kao trajne veze u poglavljiju 11. Obrazac za podnošenje je također uključen na ovim stranicama. **Primjer 13-3** pokazuje web obrazac koji će se koristiti za unos komentara— izuzetno jednostavan obrazac koji ima samo tekstualno polje i dugme za slanje.

Primjer 13-3. app/main/forms.py: obrazac za unos komentara

```
class CommentForm(FlaskForm):
    body = StringField("", validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Primjer 13-4 prikazuje ažuriranu /post/<int:id> rutu s podrškom za komentare.

Primjer 13-4. app/main/views.py: podrška za komentare na blogu

```
@main.route('/post/<int:id>', methods=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                           post=post,
                           author=current_user._get_current_object())
        db.session.add(comment)
        db.session.commit()
        flash('Vaš komentar je objavljen.')
        return redirect(url_for('.post', id=post.id, page=-1))
    page = request.args.get('page', 1, type=int)
    if stranica == -1:
        stranica = (post.comments.count() - 1) // \
            current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    paginacija = post.comments.order_by(Comment.timestamp.asc()).paginate(
        stranica, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    stranica.items.vraćaju render_template('post.html', posts=[post],
                                           form=form,
                                           komentari=komentari,
                                           paginacija=paginacija)
```

Ova funkcija prikaza instancira obrazac za komentare i šalje ga na post.html šablon za renderovanje. Logika koja ubacuje novi komentar kada se obrazac pošalje slična je rukovanju blog postovima. Kao i slučaju Post , autor komentara se ne može postaviti direktno na current_user jer je ovo proxy objekat kontekstualne varijable. Izraz current_user._get_current_object() vraća stvarni objekt korisnika .

Komentari su razvrstani po vremenskoj oznaci hronološkim redom, tako da se novi komentari uvijek dodaju na dno liste. Kada se unese novi komentar, preusmjeravanje koje završava zahtjev vraća se na isti URL, ali funkcija `url_for()` postavlja stranicu na -1, poseban broj stranice koji se koristi za traženje zadnje stranice komentara tako da upravo uneti komentar se vidi na stranici. Kada se broj stranice dobije iz niza upita i utvrdi da je -1, vrši se izračun s brojem komentara i veličinom stranice kako bi se dobio stvarni broj stranice

koristiti.

Lista komentara povezanih s objavom dobiva se putem `post.comments` odnosa jedan-prema-više, sortirana po vremenskoj oznaci komentara i paginirana istim tehnikama koje se koriste za postove na blogu. Komentari i objekt paginacije se šalju predlošku za renderiranje. Konfiguraciona varijabla `FLASKY_COMMENTS_PER_PAGE` se dodaje u `cong.py` radi kontrole veličine svake stranice komentara.

Prikaz komentara je definiran u novom predlošku, `_comments.html`, koji je sličan `_posts.html`, ali koristi drugačiji skup CSS klase. Ovaj predložak je uključen u `_posts.html` ispod tijela posta, nakon čega slijedi poziv makronaredbi za paginaciju.

Možete pregledati promjene u šablonima u GitHub spremištu aplikacije.

Da biste dovršili ovu funkciju, postovi na blogu prikazani na početnoj stranici i stranicama profila trebaju veze do stranica s komentarima. To je prikazano u **primjeru 13-5**.

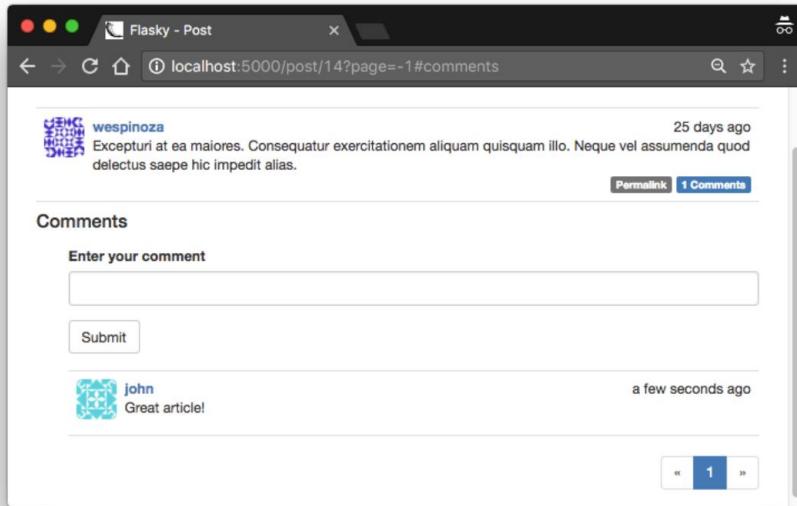
Primjer 13-5. `_app/templates/_posts.html`: link do komentara na blog postovima

```
<a href="{{ url_for('.post', id=post.id ) }}#comments"> <span
  class="label label-primary">
    {{ post.comments.count() }} Komentari </a>
```

Obratite pažnju na to kako tekst veze uključuje broj komentara, koji se lako može dobiti iz odnosa jedan-prema-više između postova i tablica komentara koristeći SQLAlchemy filter `count()`.

Zanimljiva je i struktura veze do stranice sa komentarima, koja je izgrađena kao trajna veza za objavu sa dodatkom `#comments` sufiksa. Ovaj posljednji dio naziva se URL fragment i koristi se za označavanje početne pozicije pomicanja stranice. Web pretraživač traži element sa datim ID -om i skroluje stranicu tako da se taj element pojavi na vrhu stranice. Ova početna pozicija je postavljena na naslov "Komentari" u predlošku `post.html`, koji je napisan kao `<h4 id="comments">Komentari</h4>`.

Slika 13-2 pokazuje kako se komentari pojavljuju na stranici.



Slika 13-2. Komentari objava na blogu

Dodatna promjena je napravljena u makro paginaciji. Veze za paginaciju za komentare takođe trebaju dodati #comments fragment, tako da je argument fragmenta dodat makrou i proslijeđen u pozivanju makra iz post.html šablona.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 13a da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod.

Comment Moderation

U poglavlju 9 definirana je lista korisničkih uloga, svaka sa listom dozvola. Jedna od dozvola bila je Permission.MODERATE, koja korisnicima koji ga imaju u svojim ulogama daje moć moderiranja komentara drugih.

Ova funkcija će biti izložena kao veza u navigacijskoj traci koja se pojavljuje samo korisnicima kojima je dozvoljeno da je koriste. Ovo se radi u predlošku base.html koristeći uslov, kao što je prikazano u Primeru 13-6.

Primjer 13-6. app/templates/base.html: Veza za moderiranje komentara u navigacijskoj traci

```
...
{%- if current_user.can(Permission.MODERATE) %} <li><a href="{{ url_for('main.moderate') }}>Moderiraj komentare</a></li> {%- endif %}
```

...

Stranica za moderiranje prikazuje komentare za sve objave na istoj listi, pri čemu se prvi prikazuju najnoviji komentari. Ispod svakog komentara nalazi se dugme koje može uključiti disabled atribut. /moderate ruta je prikazana u [primjeru 13-7](#).

Primjer 13-7. app/main/views.py: ruta moderiranja komentara

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE) def
moderate(): page = request.args.get('page', 1,
    type=int) pagination =
Comment.query.order_by(Comment.timestamp.desc()).paginate(page,
    per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'], error_out=False)
    comments = pagination.items return render_template('moderate.html',
    comments=comments,
    paginacija=paginacija, stranica=stranica)
```

Ovo je vrlo jednostavna funkcija koja čita stranicu komentara iz baze podataka i proslijeđuje ih šablonu za renderiranje. Zajedno s komentarima, predložak prima objekt paginacije i trenutni broj stranice.

Predložak moderate.html, prikazan u [primjeru 13-8](#), također je jednostavan jer se oslanja na podpredložak _comments.html kreiran ranije za prikazivanje comments.

Primjer 13-8. app/templates/moderate.html: šablon za moderiranje komentara

```
{% proširuje "base.html" %} {%
uvoz "_macros.html" kao makroe %}

{% block title %}Flasky - Moderacija komentara{% endblock %}

{% block page_content %}


# Moderacija komentara

 {% set moderate = Tačno %} {% if
include '_comments.html' %} {% if
paginacija %} <div class="paginacija">
```

```
    {{ macros.pagination_widget(paginacija, '.moderate') }} </div> {%
endif %} {% endblock %}
```

Ovaj predložak odgađa prikazivanje komentara predlošku _comments.html, ali prije nego što preda kontrolu podređenom predlošku koristi Ninja2-ovu set direktivu da definira umjerenu varijablu šablona postavljenu na Tačno. Ovu varijablu koristi predložak _comments.html da odredi da li je potrebno prikazati karakteristike moderiranja.

Dio predloška _comments.html koji prikazuje tijelo svakog komentara treba modifisirati na dva načina. Za obične korisnike (kada umjerena varijabla nije postavljena), svi komentari koji su označeni kao onemogućeni trebaju biti potisnuti. Za moderatora (kada je moderate postavljeno na Tačno), tijelo komentara mora biti prikazano bez obzira na deaktivirano stanje, a ispod tijela treba uključiti dugme za prebacivanje stanja. **Primjer 13-9** pokazuje ove promjene.

Primjer 13-9. app/templates/_comments.html: prikazivanje tijela komentara

```
...
<div class="comment-body">
    {% if comment.disabled %}
        <p></p><i>Ovaj komentar je onemogućio moderator.</i></p> {% endif %} {% if
    ako je umjeren ili nije comment.disabled %} {% if comment.body_html %}
        {{ comment.body_html | sigurno }} {% ostalo %} {{ comment.body }} {% endif %}
    {% endif %}
</div> {% if
moderate %} <br>
    {% if
comment.disabled %} <a
class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable') , id=comment.id,
page=page }}">Omogući</a> {% else %} <a class="btn btn-danger btn-xs"
href="{{ url_for('.moderate_disable', id=comment.id, page=page )}}>Onemogući</a>
    {% endif %}
    {% endif %}
...

```

Sa ovim promjenama, korisnici će vidjeti kratku obavijest za onemogućene komentare. Moderatori će vidjeti obavještenje i tijelo komentara. Moderatori će također vidjeti dugme za prebacivanje stanja onemogućeno ispod svakog komentara. Dugme poziva jedno od dva nova

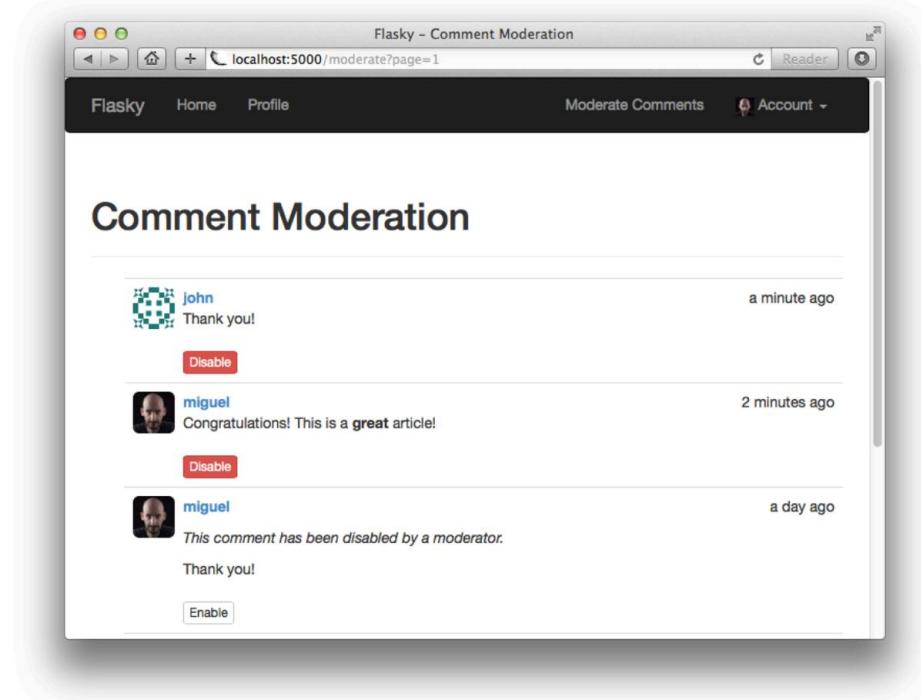
rute, ovisno o tome u koje se od dva moguća stanja komentar mijenja.
Primjer 13-10 pokazuje kako su ove rute definirane.

Primjer 13-10. app/main/views.py: rute za moderiranje komentara

```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE) def
moderate_enable(id): comment =
    Comment.query.get_or_404(id)
    comment.disabled = Netačno
    db.session.add(comment)
    return redirect(url_for('.moderate',
                           page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE) def
moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = Tačno
    db.session.add(comment)
    return redirect(url_for('.moderate',
                           page=request.args.get('page', 1, type=int)))
```

Rute za omogućavanje i onemogućavanje komentara učitavaju objekt komentara, postavljaju onemogućeno polje na odgovarajuću vrijednost i zapisuju ga nazad u bazu podataka. Na kraju se preusmjeravaju nazad na stranicu za moderiranje komentara (prikazana na [slici 13-3](#)), a ako je argument stranice dat u nizu upita, oni ga uključuju u preusmjeravanje. Dugmad u predlošku _comments.html prikazana su s argumentom stranice tako da preusmjeravanje korisnika vraća na istu stranicu.



Slika 13-3. Stranica za moderiranje komentara



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 13b da provjerite ovu verziju aplikacije.

Tema društvenih karakteristika je upotpunjena ovim poglavljem. U sljedećem poglavljju naučit ćete kako izložiti funkcionalnost aplikacije kao API koji klijenti kao što su aplikacije za pametne telefone mogu koristiti.

Sučelja za programiranje aplikacija

Posljednjih godina postoji trend u web aplikacijama da sve više i više poslovne logike premještaju na stranu klijenta, proizvodeći arhitekturu koja je poznata kao Rich Internet Applications (RIA). U RIA, glavna (a ponekad i jedina) funkcija servera je da klijentskoj aplikaciji pruži usluge preuzimanja i skladištenja podataka. U ovom modelu, server postaje web usluga ili sučelje za programiranje aplikacije (API).

Postoji nekoliko protokola pomoću kojih RIA mogu komunicirati s web uslugom. Protokoli poziva udaljenih procedura (RPC) kao što je XML-RPC ili njegov derivat, Simplified Object Access Protocol (SOAP), bili su popularan izbor prije nekoliko godina. Nedavno se arhitektura Representational State Transfer (REST) pojavila kao favorit za web aplikacije zbog toga što je izgrađena na poznatom modelu World Wide Weba.

Flask je idealan okvir za izgradnju RESTful web servisa, zahvaljujući svojoj laganoj prirodi. U ovom poglavlju ćete naučiti kako implementirati RESTful API baziran na Flask.

Uvod u REST

Doktorska disertacija Roya Fieldinga opisuje REST arhitektonski stil za web servise u smislu njegovih šest karakteristika:

Klijent-server

Mora postojati jasno razdvajanje između klijenata i servera.

Apatriid

Zahtjev klijenta mora sadržavati sve informacije koje su potrebne za njegovo izvršenje. Server ne smije pohraniti stanje o klijentu koje traje od jednog zahtjeva do drugog.

Keširani odgovori sa servera mogu biti označeni kao keširani ili ne-keširani tako da klijenti (ili posrednici između klijenata i servera) mogu koristiti keš memoriju u svrhu optimizacije.

Jedinstveni interfejs

Protokol pomoću kojeg klijenti pristupaju serverskim resursima mora biti konzistentan, dobro definisan i standardizovan. Ovo je najkompleksniji aspekt REST-a, koji pokriva upotrebu jedinstvenih identifikatora resursa, predstavljanja resursa, samoopisnih poruka između klijenta i servera i hipermehdije.

Slojeviti sistemski

proxy serveri, keš memorije ili gateway-i mogu se umetnuti između klijenata i servera po potrebi radi poboljšanja performansi, pouzdanosti i skalabilnosti.

Code-on-demand

Klijenti mogu opcionalno preuzeti kod sa servera da bi ga izvršili u svom kontekstu.

Resursi su sve Koncept resursa je

srž REST arhitektonskog stila. U ovom kontekstu, resurs je stavka od interesa u domenu aplikacije. Na primjer, u aplikaciji za bloganje, korisnici, blog postovi i komentari su svi resursi.

Svaki resurs mora imati jedinstveni identifikator koji ga predstavlja. Kada radite sa HTTP-om, identifikatori za resurse su URL-ovi. Nastavljajući s primjerom bloganja, blog post bi mogao biti predstavljen URL-om /api/posts/12345, gdje je 12345 identifikator objave, kao što je primarni ključ baze podataka posta. Format ili sadržaj URL-a zapravo nisu važni; sve što je važno je da svaki URL resursa jedinstveno identificuje resurs.

Kolekcija svih resursa u klasi također ima dodijeljen URL. URL za kolekciju blog postova može biti /api/posts/, a URL za prikupljanje svih komentara može biti /api/comments/.

API također može definirati URL-ove kolekcije koji predstavljaju logičke podskupove svih resursa u klasi. Na primjer, zbirka svih komentara u blog postu 12345 može biti predstavljena URL-om /api/posts/12345/comments/. Uobičajeno je definirati URL-ove koji predstavljaju kolekcije resursa sa kosom crtom na kraju, jer im to daje reprezentaciju "poddirektorijuma".



Imajte na umu da Flask primjenjuje poseban tretman na rute koje završavaju kosom crtom. Ako klijent zatraži URL bez završne kose crte i postoji odgovarajuća ruta koja ima kosu crtu na kraju, tada će Flask automatski odgovoriti preusmjeravanjem na URL završne kose crte. Za obrnuti slučaj ne izdaju se preusmjeravanja.

Metode zahtjeva

Klijentska aplikacija šalje zahtjeve serveru na utvrđene URL-ove resursa i koristi metodu zahtjeva da ukaže na željenu operaciju. Da bi dobio listu dostupnih blog postova u API-ju za blogovanje, klijent bi poslao GET zahtjev na `http://www.example.com/api/posts/`, a da bi umetnuo novi blog post bi poslao POST zahtjev na isti URL, sa sadržajem blog posta u tijelu zahtjeva. Da bi preuzeo blog post 12345, klijent bi poslao GET zahtjev na `http://www.example.com/api/posts/12345`. **Tabela 14-1** navodi metode zahtjeva koje se obično koriste u RESTful API-jima, sa njihovim značenjima.

Tabela 14-1. Metode HTTP zahtjeva u RESTful API-jima

Metoda zahtjeva	Target	Opis	HTTP statusni kod odgovora
GET	URL pojedinačnog resursa	Nabavite izvor.	200
GET	URL za prikupljanje resursa	Nabavite kolekciju resursa (ili jednu stranicu od nje ako server implementira paginaciju).	200
POŠTA	URL zbirke resursa	Kreirajte novi resurs i dodajte ga u kolekciju. Server bira URL novog resursa i vraća ga u zaglavljtu Lokacija u odgovoru.	201
STAVITI	URL pojedinačnog resursa	Izmjenite postojeći resurs. Alternativno, ova metoda se također može koristiti za kreiranje novog resursa kada klijent može odabrati URL resursa.	200 ili 204
IZBRIŠI	URL pojedinačnog resursa	Brisanje izvora.	200 ili 204
IZBRIŠI	URL zbirke resursa	Izbrišite sve resurse u kolekciji.	200 ili 204



REST arhitektura ne zahteva da se sve metode implementiraju za resurs. Ako klijent pozove metodu koja nije podržana za dati resurs, tada bi trebao biti vraćen odgovor sa statusnim kodom 405 (Metoda nije dozvoljena). Flask automatski obrađuje ovu grešku.

Metode zahtjeva GET, POST, PUT i DELETE nisu jedine. HTTP protokol se oslanja na druge metode, kao što su HEAD i OPTIONS, koje Flask automatski implementira.

Tijela zahtjeva i odgovora Resursi

se šalju naprijed-nazad između klijenta i servera u tijelima zahtjeva i odgovora, ali REST ne specificira format koji se koristi za kodiranje resursa. Zaglavje Content-Type u zahtjevima i odgovorima se koristi za označavanje formata u kojem je resurs kodiran u tijelu. Standardni mehanizam pregovaranja o sadržaju

Nizmi u HTTP protokolu mogu se koristiti između klijenta i servera za dogovor o formatu koji oba podržavaju.

Dva formata koja se obično koriste sa RESTful web servisima su JavaScript Object Notation (JSON) i Extensible Markup Language (XML). Za RIA-e zasnovane na webu, JSON je privlačan zbog toga što je mnogo sažetiji od XML-a i zbog bliskih veza sa JavaScriptom, skript jezikom na strani klijenta koji koriste web pretraživači.

Vraćajući se na API primjer bloga, resurs postova na blogu može imati sljedeću JSON reprezentaciju:

```
{
  "self_url": "http://www.example.com/api/posts/12345", "title": "Pisanje RESTful API-ja u Python-u", "author_url": "http://www.example.com/api/users/2", "body": "... tekst članka ovdje ...",
  "comments_url": "http://www.example.com/api/posts/12345/comments"
}
```

Obratite pažnju na to kako su polja url, author_url i comments_url potpuno kvalificirani URL-ovi resursa. Ovo je važno jer ovi URL-ovi omogućavaju klijentu da otkrije novi resurs. ove.

U dobro dizajniranom RESTful API-ju, klijent zna kratku listu URL-ova resursa najviše razine, a zatim otkriva ostatak iz veza uključenih u odgovore, slično kao što možete otkriti nove web stranice dok pretražujete web klikom na linkove koji se pojavljuju na stranicama za koje zname.

Versioniranje

U tradicionalnoj web aplikaciji usmjerenoj na server, server ima potpunu kontrolu nad aplikacijom. Kada se aplikacija ažurira, instalacija nove verzije na server je dovoljna da se ažuriraju svi korisnici jer se čak i dijelovi aplikacije koji se pokreću u korisnikovom web pretraživaču preuzimaju sa servera.

Situacija sa RIA-ima i web servisima je složenija, jer se klijenti često razvijaju nezavisno od servera — možda čak i od strane različitih ljudi. Razmotrite slučaj aplikacije u kojoj RESTful web uslugu koriste različiti klijenti uključujući web pretraživače i matične klijente pametnih telefona. Klijent web pretraživača može se ažurirati na serveru u bilo koje vrijeme, ali aplikacije za pametne telefone ne mogu se ažurirati na silu; vlasnik pametnog telefona treba da dozvoli ažuriranje. Čak i ako je vlasnik pametnog telefona voljan da se ažurira, nije moguće organizovati nadogradnju svih postojećih instanci aplikacija za pametne telefone tako da se tačno poklope sa implementacijom nove verzije servera.

Iz ovih razloga, web servisi moraju biti tolerantniji od običnih web aplikacija i biti sposobni da rade sa stariim verzijama svojih klijenata. Promjene na web servisu moraju biti urađene s krajnjom pažnjom, jer promjene koje nisu kompatibilne s prethodnim verzijama mogu uzrokovati

postojeće klijente prekinuti dok se ne nadograde. Uobičajena praksa je da se web servisima da verzija, koja se dodaje svim URL-ovima definisanim u toj verziji serverske aplikacije. Na primjer, prvo izdanje web servisa za blogovanje moglo bi otkriti kolekciju blog postova na `/api/v1/posts/`.

Uključivanje verzije web usluge u URL pomaže da stare i nove funkcije budu organizovane tako da server može pružiti nove funkcije novim klijentima, dok nastavlja da podržava stare klijente. Ažuriranje usluge bloganja moglo bi promijeniti JSON format postova na blogu i sada objaviti postove na blogu kao `/api/v2/posts/`, uz zadržavanje starijeg JSON formata za klijente koji se povezuju na `/api/v1/posts/`.

Iako podržavanje više verzija servera može postati opterećenje održavanja, postoje situacije u kojima je to jedini način da se dozvoli razvoj aplikacije bez izazivanja problema postojećim implementacijama. Starije verzije usluge mogu se odbaciti i kasnije ukloniti nakon što svi klijenti pređu na noviju verziju.

RESTful Web usluge sa Flaskom

Flask olakšava kreiranje RESTful web servisa. Poznati dekorator `route()` zajedno sa njegovim opcijskim argumentom metoda može se koristiti za deklariranje ruta koje rukuju URL-ovima resursa izloženim od strane usluge. Rad sa JSON podacima je također jednostavan, jer se JSON podaci uključeni u zahtjev mogu dobiti u formatu rječnika pozivanjem `request.get_json()`, a odgovor koji treba da sadrži JSON može se lako generirati iz Python rječnika koristeći Flask-ov `jsonify()` pomoćna funkcija.

Sljedeći odjeljci pokazuju kako se Flasky može proširiti pomoću RESTful web usluge koja klijentima daje pristup objavama na blogu i srodnim resursima.

Kreiranje API nacrta Rute

povezane sa RESTful API-jem čine samostalni podskup aplikacije, tako da je njihovo stavljanje u njihov vlastiti nacrt najbolji način da ih dobro organizirate. Opća struktura API nacrta unutar aplikacije prikazana je u [primjeru 14-1](#).

Primjer 14-1. API nacrt strukture

```

|-flasky |-
  app/ |-
    api |-
      __init__.py |-
      users.py |-
      posts.py |-
      comments.py |-
      authentication.py |-
      errors.py |decorators.py
  
```

Obratite pažnju na to kako paket koji se koristi za API uključuje broj verzije u svom nazivu. Ako u budućnosti bude potrebno uvesti verziju API-ja koja nije kompatibilna unatrag, ona se može dodati kao poseban paket s različitim brojem verzije i oba API-ja mogu biti uključena u aplikaciju.

API nacrt implementira svaki resurs u poseban modul. Uključeni su i moduli koji brinu o autentifikaciji i rukovanju greškama i za pružanje prilagođenih dekoratera. Konstruktor nacrta prikazan je u [primjeru 14-2](#).

Primjer 14-2. app/api/__init__.py: Kreiranje API nacrta

```
iz flask import Blueprint

api = Nacrt('api', __name__)

od . autentifikaciju uvoza , postove, korisnike, komentare, greške
```

Struktura konstruktora paketa nacrta slična je strukturi ostalih nacrta. Uvoz svih komponenti nacrta je neophodan kako bi se registrovale rute i drugi rukovaoci. Budući da mnogi od ovih modula moraju da uvezu nacrt API-ja koji je ovdje referenciran, uvozi se obavljaju na dnu kako bi se sprječile greške zbog kružnih ovisnosti.

Registracija API nacrta je prikazana u [primjeru 14-3](#).

Primjer 14-3. app/init.py: registracija API nacrta

```
def create_app(config_name):
    #
    iz .api import api kao api_blueprint
    app.register_blueprint(api_blueprint, url_prefix='/api/v1') #
    ...
```

API nacrt se registruje sa URL prefiksom, tako da će sve njegove rute imati svoje URL adrese sa prefiksom /api/v1. Dodavanje prefiksa prilikom registracije nacrta je dobra ideja jer eliminira potrebu za tvrdim kodiranjem broja verzije u svakom nacrtu ruta.

Rukovanje

greškama RESTful web usluga obavještava klijenta o statusu zahtjeva slanjem odgovarajućeg HTTP statusnog koda u odgovoru, plus sve dodatne informacije u tijelu odgovora. Tipični statusni kodovi koje klijent može očekivati od web servisa navedeni su u [Tabeli 14-2](#).

Tabela 14-2. HTTP kodovi statusa odgovora koje obično vraćaju API-ji

HTTP statusni kod	Ime	Opis
200	uredu	Zahtjev je uspješno završen.
201	Created	Zahtjev je uspješno dovršen i kao rezultat je kreiran novi resurs.
202	Prihvaćeno	Zahtjev je prihvaćen za obradu, ali je još uvjek u toku i izvoditi će se asinhrono.
204	Nema sadržaja	Zahtjev je uspješno završen i u odgovoru nema podataka za vraćanje.
400	Loš zahtjev	Zahtjev je nevažeći ili nedosljedan.
401	Neovlašteno	Zahtjev ne uključuje informacije o autentifikaciji ili su navedeni vjerodajnici nevažeći.
403	Zabranjeno	Akreditivi za provjeru autentičnosti poslani sa zahtjevom nisu dovoljni za zahtjev.
404	Nije pronađeno	Resurs naveden u URL-u nije pronađen.
405	Metoda nije dozvoljena	Tražena metoda nije podržana za dati resurs.
500	Interna greška servera	Došlo je do neočekivane greške prilikom obrade zahtjeva.

Rukovanje statusnim kodovima 404 i 500 predstavlja malu komplikaciju, jer ove greške obično generiše sam Flask i vraća HTML odgovor. Ovo može zbuniti API klijenta, koji će vjerovatno očekivati sve odgovore u JSON formatu.

Jedan od načina da se generišu odgovarajući odgovori za sve klijente je da se nateraju rukovaoci grešaka da prilagode svoje odgovore na osnovu formata koji zahteva klijent, tehnika koja se zove pregovaranje o sadržaju. **Primjer 14-4** pokazuje poboljšani rukovalac greškama 404 koji odgovara sa JSON klijentima web usluga i HTML-om za druge. Rukovalac greškama 500 je napisan na sličan način.

Primjer 14-4. app/api/errors.py: 404 obrađivač grešaka sa HTTP pregovaranjem sadržaja

```
@main.app_errorhandler(404)
def page_not_found(e):
    ako request.accept_mimetypes.accept_json i \
        ne request.accept_mimetypes.accept_html:
        odgovor = jsonify({'error': 'nije pronađen'})
        response.status_code = 404 povratak odgovor return
    render_template('404.html'), 404
```

Ova nova verzija rukovaoca greškama provjerava zaglavje zahtjeva za prihvaćanje, koje je dekodirano u `request.accept_mimetypes`, kako bi odredilo u kojem formatu klijent želi odgovor. Preglednici općenito ne specificiraju nikakva ograničenja za formate odgovora, ali API klijenti obično to rade. JSON odgovor se generira samo za klijente koji uključuju JSON u svoju listu prihvaćenih formata, ali ne i HTML.

Preostale statusne kodove generira eksplisitno web servis, tako da se mogu implementirati kao pomoćne funkcije unutar nacrtu u modulu errors.py.

Primer 14-5 pokazuje implementaciju greške 403; ostali su slični.

Primjer 14-5. app/api/errors.py: API za rukovanje greškama za statusni kod 403

```
def zabranjeno(poruka):
    response = jsonify({'error': 'zabranjeno', 'message': poruka})
    response.status_code = 403 povratni odgovor
```

Funkcije prikaza u nacrtu API-ja mogu pozvati ove pomoćne funkcije za generiranje odgovora na greške kada je to potrebno.

Autentifikacija korisnika pomoću Flask-HTTPAuth

Web usluge, poput običnih web aplikacija, moraju zaštитiti informacije i osigurati da se one ne daju neovlaštenim stranama. Iz tog razloga, RIA moraju tražiti od svojih korisnika vjerodajnjice za prijavu i proslijediti ih serveru radi provjere.

Ranije je spomenuto da je jedna od karakteristika RESTful web servisa to što su bez stanja, što znači da serveru nije dozvoljeno da „zapamti“ bilo šta o klijentu između zahtjeva. Klijenti moraju u samom zahtevu da navedu sve informacije potrebne za izvršenje zahteva, tako da svi zahtevi moraju sadržati korisničke akredititive.

Trenutna funkcionalnost prijave implementirana uz pomoć Flask-Login-a pohranjuje podatke u korisničkoj sesiji, koje Flask po defaultu pohranjuje u kolačić na strani klijenta, tako da server ne pohranjuje nikakve informacije vezane za korisnika; umjesto toga traži od klijenta da ga pohrani. Čini se da je ova implementacija u skladu sa zahtjevom REST-a bez državljanstva, ali korištenje kolačića u RESTful web servisima spada u sivu zonu, jer može biti glomazno za klijente koji nisu web pretraživači da ih implementiraju. Iz tog razloga, korištenje kolačića u API-jima općenito se smatra lošim izborom dizajna.



Zahtjev bez državljanstva REST-a može izgledati prestrog, ali nije proizvoljan. Serveri bez stanja mogu se vrlo lako skalirati. Ako serveri pohranjuju informacije o klijentima, potrebno je osigurati da isti server uvijek prima zahtjeve od datog klijenta, ili u suprotnom koristiti zajedničku pohranu za podatke klijenta. Oba su složeni problemi za rješavanje koji ne postoje kada je server bez državljanstva.

Budući da je RESTful arhitektura zasnovana na HTTP protokolu, HTTP autentifikacija je preferirani metod koji se koristi za slanje akreditiva, bilo u osnovnom ili sažetom obliku. Uz HTTP autentifikaciju, korisničke vjerodajnjice su uključene u zaglavje autorizacije sa svim zahtjevima.

HTTP protokol autentikacije je dovoljno jednostavan da se može direktno implementirati, ali proširenje Flask-HTTPAuth pruža zgodan omot koji skriva detalje protokola u dekoratoru sličnom Flask-Login-ovom `login_required`.

Flask-HTTPAuth se instalira sa pip-om:

```
(venv) $ pip install flask-httpauth
```

Da biste inicijalizirali ekstenziju za HTTP Basic autentifikaciju, mora se kreirati objekt klase `HTTPBasicAuth`. Kao i Flask-Login, Flask-HTTPAuth ne pravi nikakve pretpostavke o proceduri koja je potrebna za provjeru korisničkih akreditiva, tako da se ove informacije daju u funkciji povratnog poziva. [Primjer 14-6](#) pokazuje kako se ekstenzija inicijalizira i pruža povratni poziv za verifikaciju.

[Primjer 14-6. app/api/authentication.py: Flask-HTTPAuth inicijalizacija](#)

```
from flask_httpauth import HTTPBasicAuth auth
= HTTPBasicAuth()

@auth.verify_password
def verify_password(e-pošta, lozinka): ako
    email == "": vrati False user =
        User.query.filter_by(email =
    email).first() ako nije korisnik:
        return False
    g.current_user = korisnik
    vrati user.verify_password(password)
```

Budući da će se ovaj tip provjere autentičnosti korisnika koristiti samo u nacrtu API-ja, ekstenzija Flask-HTTPAuth se inicijalizira u paketu nacrtu, a ne u paketu aplikacije kao druge ekstenzije.

E-mail i lozinka se provjeravaju korištenjem postojeće podrške u korisničkom modelu. Povratni poziv za provjeru vraća `True` kada je prijava važeća i `False` u suprotnom. Ekstenzija Flask-HTTPAuth će također pozvati povratni poziv za zahtjeve koji ne nose autentifikaciju, postavljajući oba argumenta na prazan niz. U ovom slučaju, kada je email prazan niz, funkcija odmah vraća `False` da blokira zahtjev; za određene aplikacije može biti prihvatljivo dozvoliti anonimnom korisniku vraćanjem `True`. Povratni poziv za autentifikaciju spremi autenticiranog korisnika u Flask-ovoј g kontekstualnoj varijabli tako da joj funkcija pregleda može pristupiti kasnije.



Budući da se korisnički akreditivi razmjenjuju sa svakim zahtjevom, izuzetno je važno da API rute budu izložene preko sigurnog HTTP-a kako bi svi zahtjevi i odgovori bili šifrirani u prijenosu.

Kada su vjerodajnice za provjelu autentičnosti nevažeće, poslužitelj vraća klijentu odgovor 401 statusnog koda. Flask-HTTPAuth generira odgovor sa ovim statusnim kodom prema zadanim postavkama, ali kako bi se osiguralo da je odgovor konzistentan s drugim greškama koje vraća API, odgovor na grešku se može prilagoditi kao što je prikazano u [primjeru 14-7](#).

Primjer 14-7. `_app/api/authentication.py`: Obrađivač grešaka Flask-HTTPAuth

```
iz .errors uvoz neovlašten
```

```
@auth.error_handler
def auth_error(): vrati
    neovlašteno('Nevažeće vjerodajnice')
```

Za zaštitu rute koristi se dekorator `auth.login_required` :

```
@api.route('/posts/')
@auth.login_required def
get_posts(): prolaz
```

Ali pošto sve rute u nacrtu moraju biti zaštićene na isti način, dekorator `login_required` može se jednom uključiti u obrađivač `before_request` za nacrt, kao što je prikazano u [primjeru 14-8](#).

Primjer 14-8. `app/api/authentication.py`: obrađivač `before_request` s autentifikacijom

```
iz .errors uvoz zabranjen
```

```
@api.before_request
@auth.login_required
def before_request(): ako
    nije g.current_user.is_anonymous i \ nije
        g.current_user.confirmed:
    return forbidden('Nepotvrđeni račun')
```

Sada će se provjere autentičnosti obaviti automatski za sve rute u nacrtu. Kao dodatna provjera, obrađivač `before_request` također odbija autentifikovane korisnike koji nisu potvrdili svoje naloge.

Autentifikacija zasnovana na tokenima

Klijenti moraju poslati vjerodajnice za autentifikaciju uz svaki zahtjev. Da biste izbegli konstantan prenos osetljivih informacija kao što je lozinka, može se koristiti rešenje za autentifikaciju zasnovano na tokenu.

U autentifikaciji zasnovanoj na tokenu, klijent zahtjeva pristupni token slanjem zahtjeva koji uključuje vjerodajnice za prijavu kao autentifikaciju. Token se tada može koristiti umjesto akreditiva za prijavu za provjeru autentičnosti zahtjeva. Iz sigurnosnih razloga,

tokeni se izdaju s pripadajućim istekom. Kada istekne token, klijent se mora ponovo autentifikovati da bi dobio novi. Rizik da žeton dođe u pogrešne ruke je ograničen zbog njegovog kratkog vijeka trajanja. **Primjer 14-9** pokazuje dvije nove metode dodane modelu korisnika koje podržavaju generiranje i verifikaciju tokena za autentifikaciju korištenjem njegovog opasnog.

Primjer 14-9. app/models.py: podrška za autentifikaciju zasnovanu na tokenu

```
class User(db.Model): #
    def.._
        generate_auth_token(self, expiration):
            s = serijalizator(current_app.config['SECRET_KEY'],
                              expires_in=istek)
            return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token): s =
        Serializator(current_app.config['SECRET_KEY']) pokušaj:
        podaci = s.loads(token)

    osim:
        vrati Ništa
        vrati User.query.get(podaci['id'])
```

Generirati_auth_token() metoda vraća potpisani token koji kodira polje id korisnika . Koristi se i vrijeme isteka u sekundama. Metoda verify_auth_token() uzima token i, ako se utvrdi da je ispravan, vraća korisnika pohranjenog u njemu. Ovo je statična metoda, jer će korisnik biti poznat tek nakon što se token dekodira.

Za provjeru autentičnosti zahtjeva koji dolaze s tokenom, verify_password povratni poziv za Flask-HTTPAuth mora biti modificiran da prihvati tokene kao i redovne vjerodajnice.

Ažurirani povratni poziv je prikazan u **primjeru 14-10**.

Primjer 14-10. app/api/authentication.py: poboljšana provjera autentičnosti s podrškom za tokene

```
@auth.verify_password
def verify_password(email_or_token, lozinka):
    ako email_or_token == "":
        return False if
    password == "":
        g.current_user = User.verify_auth_token(email_or_token) g.token_used
        = Tačan povratak g.current_user nije None user =
        User.query.filter_by(email=email_or_token).first() ako nije korisnik . :

    return False
    g.current_user = korisnik
```

```
g.token_used = Lažni
povratak user.verify_password(password)
```

U ovoj novoj verziji, prvi argument za autentifikaciju može biti adresa e-pošte ili token za autentifikaciju. Ako je ovo polje prazno, prepostavlja se da je anonimni korisnik, kao i ranije. Ako je lozinka prazna, tada se prepostavlja da je polje email_or_token token i kao takvo je potvrđeno. Ako oba polja nisu prazna, onda se prepostavlja regularna autentifikacija putem e-pošte i lozinke. Sa ovom implementacijom, autentifikacija zasnovana na tokenu je opcionalna; na svakom klijentu je da ga koristi ili ne. Da bi se funkcijama prikaza dala mogućnost razlikovanja između dva metoda provjere autentičnosti dodana je varijabla g.token_used.

Ruta koja vraća tokene za autentifikaciju klijentu je također dodana nacrtu API-ja. Implementacija je prikazana u [primjeru 14-11](#).

Primjer 14-11. app/api/authentication.py: generiranje tokena za autentifikaciju

```
@api.route('/tokens/', methods=['POST'])
def get_token():
    ako je g.current_user.is_anonymous
        ili g.token_used: return unauthorized ('Nevažeće
                                             vjerodajnice')
    return jsonify({'token':
        g.current_user.generate_auth_token(
            expiration=3600), 'expiration': 3600})
```

Pošto je ova ruta u nacrtu, mehanizmi provjere autentičnosti dodani rukovaocu before_request također se primjenjuju na nju. Kako bi sprječili klijente da se autentifikuju na ovoj ruti koristeći prethodno dobijeni token umjesto adrese e-pošte i lozinke, provjerava se varijabla g.token_used, a zahtjevi autentificirani tokenom se odbijaju. Svrha ovoga je sprječiti korisnike da zaobiđu rok trajanja tokena tražeći novi token koristeći stari token kao autentifikaciju. Funkcija vraća token u JSON odgovoru s periodom valjanosti od jednog sata. Tačka je također uključena u JSON odgovor.

Serijsalizacija resursa u i iz JSON- a Česta potreba

prilikom pisanja web usluge je pretvaranje internih reprezentacija resursa u i iz JSON-a, koji je transportni format koji se koristi u HTTP zahtjevima i odgovorima. Proces pretvaranja interne reprezentacije u transportni format kao što je JSON naziva se serijsalizacija. [Primjer 14-12](#) pokazuje novu metodu to_json() dodanu klasi Post .

Primjer 14-12. app/models.py: pretvaranje objave u JSON serijalizirajući rječnik

```
klasa Post(db.Model):
    # ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id), 'body': self.body,
            'body_html': self.body_html, 'timestamp':
            self.timestamp, 'author_url': url_for('api.get_user',
            id=self.author_id), 'comments_url':
            url_for('api.get_post_comments', id=self.id), 'comment_count':
            self.comments.count()

        }
        return json_post
```

Polja url, author_url i comments_url moraju vratiti URL-ove za odgovarajuće resurse, tako da se generiraju pozivima url_for() na druge rute koje će biti definirane u nacrtu API-ja.

Ovaj primjer pokazuje kako je moguće vratiti „izmišljene“ atribute u predstavljanju resursa. Polje comment_count vraća broj komentara koji postoji za post na blogu. Iako ovo nije stvarni atribut modela, on je uključen u reprezentaciju resursa kao pogodnost za klijenta.

Metoda to_json() za korisničke modele može se konstruirati na sličan način. Ova metoda je prikazana u [primjeru 14-13](#).

Primjer 14-13. app/models.py: pretvaranje korisnika u JSON serijalizirajući rječnik

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_user', id=self.id), 'username':
            self.username, 'member_since': self.member_since,
            'last_seen': self.last_seen, 'posts_url': url_for(
            'api.get_user_posts', id=self.id), 'followed_posts_url':
            url_for('api.get_user_followed_posts', id=self.id), 'post_count':
            self.posts.count()

        }
        vratи json_user
```

Obratite pažnju na to kako su u ovoj metodi neki od atributa korisnika, kao što su e-pošta i uloga, izostavljeni iz odgovora iz razloga privatnosti. Ovaj primjer još jednom demonstrira

da reprezentacija resursa koji se nudi klijentima ne mora biti identična internoj definiciji odgovarajućeg modela baze podataka.

Inverznost serijalizacije naziva se deserijalizacija. Deserijalizacija JSON strukture nazad u model predstavlja izazov da neki podaci koji dolaze od klijenta mogu biti nevažeći, pogrešni ili nepotrebni. **Primer 14-14** pokazuje metodu koja kreira post iz JSON-a.

Primjer 14-14. app/models.py: kreiranje objave na blogu iz JSON-a

```
iz app.exceptions import ValidationError

class Post(db.Model): #
    @staticmethod def
        from_json(json_post):
            body = json_post.get('body')
            ako je tijelo Ništa ili tijelo == "":
                podizanje ValidationError('objava
                    nema tijelo') povratni post(body=body)
```

Kao što vidite, ova implementacija bira da koristi samo atribut tijela iz JSON rječnika. Atribut body_html se zanemaruje budući da se renderiranje Markdown na strani servera automatski pokreće SQLAlchemy događajem kad god se modifcira atribut tijela . Atribut vremenske oznake se ne mora davati osim ako klijentu nije dozvoljeno objavljivanje postova unatrag ili budućih datuma, što nije funkcija koju ova aplikacija podržava. Polje author_url se ne koristi jer klijent nema ovlaštenje da izabere autora blog posta; jedina moguća vrijednost za ovo polje je vrijednost autentificiranog korisnika. Atributi comments_url i comment_count se automatski generišu iz odnosa baze podataka, tako da u njima nema korisnih informacija koje su potrebne za kreiranje objave. Konačno, polje url se zanemaruje jer u ovoj implementaciji URL-ove resursa definira server, a ne klijent.

Obratite pažnju na to kako se vrši provjera grešaka. Ako polje tijela nedostaje ili je prazno tada se pokreće izuzetak ValidationError . Pokretanje izuzetka je u ovom slučaju prikladan način da se pozabavite greškom jer ova metoda nema dovoljno znanja da pravilno obradi stanje greške. Izuzetak efektivno prosljeđuje grešku pozivaocu, omogućavajući kodu višeg nivoa da obradi grešku. Klasa ValidationError implementirana je kao jednostavna potklasa Pythonove ValueError.

Ova implementacija je prikazana u **primjeru 14-15**.

Primjer 14-15. app/exceptions.py: ValidationError izuzetak

```
klasa ValidationError(ValueError):
    pass
```

Aplikacija sada treba da obradi ovaj izuzetak pružanjem odgovarajućeg odgovora klijentu. Da biste izbegli dodavanje koda za hvatanje izuzetaka u funkcije prikaza, globalni rukovalac izuzetkom se može instalirati pomoću Flaskovog dekoratera za obradu grešaka. Rukovalac za izuzetak ValidationError prikazan je u [primjeru 14-16](#).

Primjer 14-16. app/api/errors.py: API rukovalac greškama za ValidationError izuzetke

```
@api.errorhandler(ValidationError) def
validation_error(e): vratи
    bad_request(e.args[0])
```

Dekorator za obradu grešaka je isti onaj koji se koristi za registraciju rukovalaca za HTTP statusne kodove, ali u ovoj upotrebi uzima klasu Exception kao argument. Dekorirana funkcija će biti pozvana svaki put kada se podigne izuzetak date klase.

Imajte na umu da je dekorater dobijen iz API nacrta, tako da će ovaj rukovalac biti pozvan samo kada se podigne izuzetak dok se rukuje rutom iz nacrta. Koristeći ovu tehniku, kod u funkciji prikaza može se napisati vrlo jasno i koncizno, bez potrebe za provjerom grešaka. Na primjer:

```
@api.route('/posts/', methods=['POST']) def
new_post(): post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post) db.session.commit()
    return jsonify(post.to_json())
```

Implementacija krajnjih tačaka resursa Ono

Što ostaje je implementacija ruta koje rukuju različitim resursima. GET zahtjevi su obično najjednostavniji jer samo vraćaju informacije i ne moraju unositi nikakve promjene. [Primer 14-17](#) prikazuje dva GET rukovaoca za blog postove.

Primjer 14-17. app/api/posts.py: GET rukovaoce resursima za postove

```
@api.route('/posts/') def
get_posts(): postovi =
    Post.query.all() return
    jsonify({ 'objave': [post.to_json() za postove u objavama] })

@api.route('/posts/<int:id>') def
get_post(id): post =
    Post.query.get_or_404(id) return
    jsonify(post.to_json())
```

Prva ruta obrađuje zahtjev za prikupljanje postova. Ova funkcija koristi razumijevanje liste za generiranje JSON verzije svih postova. Druga ruta

vraća jednu objavu na blogu i odgovara greškom koda 404 kada dati ID nije pronađen u bazi podataka.

POST rukovalac za resurse blog postova umeće novi blog post u bazu podataka. Ova ruta je prikazana u [primjeru 14-18](#).

Primjer 14-18. app/api/posts.py: POST obrađivač resursa za postove

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE) def
new_post(): post = Post.from_json(request.json)
post.author = g.current_user db.
session.add(post) db.session.commit()
return jsonify(post.to_json()), 201,
{'Location': url_for('api.get_post', id=post.id)}
```

Ova funkcija prikaza je umotana u permission_required dekorator (prikazano u narednom primjeru) koji osigurava da autentificirani korisnik ima dozvolu za pisanje blog postova. Stvarno kreiranje blog posta je jednostavno zbog podrške za rukovanje greškama koja je prethodno implementirana. Objava na blogu se kreira iz JSON podataka i njegov autor je eksplicitno dodijeljen kao ovlašteni korisnik. Nakon što se model upiše u bazu podataka, vraća se statusni kod 201 i dodaje se zaglavljne lokacije s URL-om novokreiranog resursa.

Imajte na umu da kao pogodnost za klijente tijelo odgovora uključuje novi resurs. Ovo će spasiti klijenta od potrebe da za njega izda GET zahtjev odmah nakon kreiranja resursa.

Permission_required dekorater koji se koristi za sprečavanje neovlašćenih korisnika da kreiraju nove postove na blogu sličan je onom koji se koristi u aplikaciji, ali je prilagođen za nacrt API-ja. Implementacija je prikazana u [primjeru 14-19](#).

Primjer 14-19. app/api/decorators.py: permission_required dekorater

```
def permission_required(permission): def
dekorator(f): @wraps(f) def
decorated_function(*args,
**kwargs):
ako nije g.current_user.can(permission): return
forbidden('Nedovoljne dozvole')
return f(*args, **kwargs)
return decorated_function
return dekorator
```

PUT obrađivač za blog postove, koji se koristi za uređivanje postojećih resursa, prikazan je u [primjeru 14-20](#).

Primjer 14-20. app/api/posts.py: PUT obrađivač resursa za postove

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE) def edit_post(id):
    post = Post.query.get_or_404(id) ako je g. trenutni_user != None
        post.author i \ ne g.current_user.can(Permission.ADMIN):
            return forbidden('Nedovoljne dozvole') post.body =
                request.json.get ('body', post.body) db.session.
                    add(post) db.session.commit() return jsonify(post.to_json())
```

Provjere dozvola su u ovom slučaju složenije. Standardna provjera dozvole za pisanje postova na blogu se vrši pomoću dekoratora, ali da bi se omogućilo korisniku da uređuje blog post, funkcija također mora osigurati da je korisnik autor objave ili je administrator. Ova provjera se eksplicitno dodaje funkciji pogleda. Ako se ova provjera mora dodati u mnoge funkcije prikaza, izgradnja dekoratora za to bi bio dobar način da se izbjegne ponavljanje koda.

Budući da aplikacija ne dozvoljava brisanje postova, rukovalac za metodu zahtjeva DELETE ne mora biti implementiran.

Obrađivači resursa za korisnike i komentare implementirani su na sličan način.

[Tabela 14-3](#) navodi skup resursa implementiranih za ovu aplikaciju i HTTP metode koje svaka podržava. Kompletan implementaciju je dostupna za proučavanje u GitHub repozitorijumu za ovu aplikaciju.

Tabela 14-3. Flasky API resursi

URL resursa	Metod	Opis	Vratite
/users/<int:id>	GET		korisnika.
/users/<int:id>/posts/	GET		Vrati sve postove na blogu koje je napisao korisnik.
/users/<int:id>/timeline/	GET		Vrati sve postove na blogu koje prati korisnik.
/objave/ /	GET		Vrati sve postove na blogu.
objave/	POST	Kreirajte novi blog post.	
/posts/<int:id> /	GET		Vrati post na blogu.
posts/<int:id>	STAVITI		Izmjenite post na blogu.
/posts/<int:id>/comments/	GET		Vratite komentare na blog post.
/posts/<int:id>/comments/	POST	Dodajte komentar na blog post.	
/komentari/	GET		Vrati sve komentare.

URL resursa	Opis metode	
/comments/<int:id>	GET	Vrati komentar.

Imajte na umu da implementirani resursi nude samo podskup funkcionalnosti koji je dostupan putem web aplikacije. Lista podržanih resursa može se proširiti ako je potrebno, kao što je otkrivanje pratilaca, omogućavanje moderiranja komentara i implementacija bilo koje druge funkcije koje bi mogle trebati API klijentu.

Paginacija velikih kolekcija resursa GET zahtjevi koji vraćaju kolekciju resursa mogu biti izuzetno skupi i teški za upravljanje za veoma velike kolekcije. Kao i web aplikacije, web servisi mogu odabrati paginaciju kolekcija.

Primer 14-21 pokazuje moguću implementaciju paginacije za listu blog postova.

Primjer 14-21. app/api/posts.py: Paginacija objave

```
@api.route('/posts/') def
get_posts(): page =
    request.args.get('page', 1, type=int) pagination =
    Post.query.paginate( page,
        per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
    postovi = pagination.items prev = Nema ako pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1)
        sljedeći = Ništa
        ako pagination.has_next:
            next = url_for('api.get_posts', page=page+1) return
            jsonify({ 'posts': [post.to_json() za post u postovima],
                'prev_url': prev, 'next_url': next, 'count': paginacija.total
            })
    }
```

Polje postova u JSON odgovoru sadrži stavke podataka kao i prije, ali sada je to samo stranica, a ne kompletan set. Stavke prev_url i next_url sadrže URL-ove resursa za prethodnu i sljedeće stranice, ili None kada stranica u tom smjeru nije dostupna. Vrijednost brojanja je ukupan broj artikala u kolekciji.

cija.

Ova tehnika se može primijeniti na sve rute koje vraćaju kolekcije.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 14a da provjerite ovu verziju aplikacije. Da biste bili sigurni da imate instalirane sve zavisnosti, također pokrenite pip install -r requirements/dev.txt.

Testiranje web servisa sa HTTPie-om

Za testiranje web usluge mora se koristiti HTTP klijent. Dva najčešće korišćena klijenta za testiranje Python web servisa iz komandne linije su cURL i HTTPie. Iako su oba korisni alati, potonji ima mnogo sažetiju i čitljiviju sintaksu komandne linije koja je posebno prilagođena API zahtjevima. HTTPie je instaliran sa pip-om:

```
(venv) $ pip install httpie
```

Pod pretpostavkom da razvojni server radi na podrazumevanoj adresi `http://127.0.0.1:5000`, GET zahtev se može izdati iz drugog prozora terminala na sledeći način:

```
(venv) $ http --json --auth <e-pošta>:<lozinka> GET \> http://127.0.0.1:5000/api/v1/posts HTTP/1.0 200 OK Dužina sadržaja: 7018 Vrsta sadržaja: application/json Datum: Sun, 22 Dec 2013 08:11:24 GMT Server: Werkzeug/0.9.4 Python/2.7.3
```

```
{
    "objave": [
        ...
    ],
    "prev_url": null
    "next_url": "http://127.0.0.1:5000/api/v1/posts/?page=2", "count": 150
}
```

Obratite pažnju na veze za paginaciju uključene u odgovor. Pošto je ovo prva stranica, prethodna stranica nije definisana, ali je vraćen URL za dobijanje sledeće stranice i ukupan broj.

Slijedeća naredba šalje POST zahtjev za dodavanje novog blog posta:

```
(venv) $ http --auth <e-pošta>:<lozinka> --json POST \> http://127.0.0.1:5000/api/v1/posts/ \> "body=dodajem objavu iz *komandna linija*"
HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json
Datum: ned, 22. decembar 2013. 08:30:27
GMT Lokacija: http://127.0.0.1:5000/api/v1/posts/111
Server : Werkzeug/0.9.4 Python/2.7.3
```

```
{
```

```

    "author": "http://127.0.0.1:5000/api/v1/users/1", "body":  

    "Dodajem objavu iz *komandne linije*.", "body_html": "<  

    p>Dodajem objavu iz <em>komandne linije</em>.</p>", "comments": "http://  

    127.0.0.1:5000/api/v1/posts/111/comments ", "comment_count": 0, "timestamp": "Ned,  

    22. decembar 2013. 08:30:27 GMT", "url": "http://127.0.0.1:5000/api/v1/posts/111"  

    }
  
```

Za korištenje tokena za autentifikaciju umjesto korisničkog imena i lozinke, prvo se šalje POST zahtjev na /api/v1/tokens/:

```
(venv) $ http --auth <e-pošta>:<lozinka> -json POST \> http://  

127.0.0.1:5000/api/v1/tokens/ HTTP/1.0 200 OK Dužina sadržaja:  

162 Vrsta sadržaja : application/json Datum: Sub, 04 Jan 2014  

08:38:47 GMT Server: Werkzeug/0.9.4 Python/3.3.3
```

```
{
  "expiration": 3600,
  "token": "eyJpYXQiOjEzODg4MjQ3MjcsImV4cCI6MTM4ODgyODMyNywiYWxnIjoiSFMy..."
```

A sada se vraćeni token može koristiti za upućivanje poziva u API sljedećih sat vremena tako što ćete ga proslijediti u polje korisničkog imena i ostaviti lozinku praznu:

```
(venv) $ http -json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1/posts/
```

Kada token istekne, zahtjevi će biti vraćeni s greškom koda 401, što ukazuje da je potrebno nabaviti novi token.

Čestitamo! Ovo poglavlje završava drugi dio, a time je završena faza razvoja karakteristika Flaskyja. Sljedeći korak je očito njegovo postavljanje, a to donosi novi skup izazova koji su predmet III dijela.

DIO III

Poslednja milja

POGLAVLJE 15

Testiranje

Postoje dva vrlo dobra razloga za pisanje jediničnih testova. Prilikom implementacije nove funkcionalnosti, jedinični testovi se koriste kako bi se potvrdilo da novi kod radi na očekivani način. Isti rezultat se može dobiti i ručnim testiranjem, ali naravno automatizirani testovi štede vrijeme i trud jer se mogu lako ponoviti.

Drugi, važniji razlog je taj što svaki put kada se aplikacija modificira, svi testovi jedinica izgrađeni oko nje mogu biti izvršeni kako bi se osiguralo da nema regresije u postojećem kodu; drugim riječima, da nove promjene nisu uticale na način na koji stari kod radi.

Jedinični testovi su deo Flaskyja od samog početka, sa testovima dizajniranim da vežbaju specifične karakteristike aplikacije implementirane u klasama modela baze podataka. Ove klase je lako testirati izvan konteksta pokrenute aplikacije, pa s obzirom na to da je potrebno malo truda, implementacija jediničnih testova za sve karakteristike koje postoje u modelima baze podataka je najbolji način da se osigura da se barem taj dio aplikacije pokrene robustan i takav ostaje.

Ovo poglavlje govori o načinima poboljšanja i proširenja testiranja jedinica na druga područja primjene.

Pribavljanje izvještaja o pokrivenosti koda

Posjedovanje testnog paketa je važno, ali je jednako važno znati koliko je dobar ili loš. Alati za pokrivanje koda mjere koliko se aplikacije koristi jediničnim testovima i mogu pružiti detaljan izvještaj koji pokazuje koji dijelovi koda aplikacije se ne testiraju. Ove informacije su od neprocjenjive vrijednosti, jer se mogu koristiti za usmjeravanje napora pisanja novih testova na područja kojima je to najpotrebniјe.

Python ima odličan alat za pokrivanje koda koji se prikladno naziva pokrivenost. Možete ga instalirati pomoću pip-a:

```
(venv) $ pip pokrivenost instalacije
```

Ovaj alat dolazi kao skripta komandne linije koja može pokrenuti bilo koju Python aplikaciju s omogućenom pokrivenošću koda, ali također pruža praktičniji pristup skripti za programsko pokretanje motora pokrivenosti. Da bi metrika pokrivenosti bila lepo integrisana u komandu flask test dodanu u [Poglavlju 7](#), može se dodati opcija --coverage . Implementacija ove opcije je prikazana u [primjeru 15-1](#).

Primjer 15-1. flasky.py: metrika pokrivenosti

```
import os
import sys
import kliknite

COV = Nema
if os.environ.get('FLASK_COVERAGE'):
    uvoz pokrivenosti
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

# ...

@app.cli.command()
@click.option('--coverage/--no-coverage', default=False, help='Pokreni
    testove pod pokrivenošću koda.')
def test(pokrivanje):

    """Pokreni jedinične testove."""
    ako pokrivenost a ne os.environ.get('FLASK_COVERAGE'):
        os.environ['FLASK_COVERAGE'] = '1' os.execvp(sys.executable,
            [sys.executable] + sys.argv) import unittest testovi =
        unittest .TestLoader().discover('testovi')
        unittest.TextTestRunner(verbosity=2).run(testovi) ako COV: COV.stop()

    COV.save()
    print('Sažetak pokrivenosti:')
    COV.report()
    basedir = os.path.abspath(os.path.dirname(__file__)) covdir =
        os.path.join(basedir, 'tmp/coverage')
    COV.html_report(directory=covdir)
    print('HTML verzija: file:///%s/index.html' % covdir)
    COV.erase()
```

Podrška pokrivenosti koda je omogućena proslijeđivanjem opcije --coverage komandi flask test . Da biste dodali Boolean opciju u test prilagođenu komandu,

`click.option` dekorator se koristi. Kliknite zatim prosjeđuje vrijednost Booleove zastave kao argument funkciji.

Ali integriranje pokrivenosti koda u `flasky.py` skriptu predstavlja mali problem. U trenutku kada je opcija `--coverage` primljena u `test()` funkciji, već je prekasno da se omogući metrika pokrivenosti; do tada je sav kod u globalnom opsegu već izvršen. Dakle, da bi dobili tačne metrike, skripta se rekurzivno ponovo pokreće nakon postavljanja varijable okruženja `FLASK_COVERAGE`. U drugom pokretanju, vrh skripte otkriva da je varijabla okruženja postavljena i uključuje pokrivenost od početka, čak i prije uvoza svih aplikacija.

Funkcija `coverage.coverage()` pokreće motor pokrivenosti. Opcija `branch=True` omogućava analizu pokrivenosti grane, koja, pored praćenja koje se linije koda izvršavaju, provjerava da li su za svaki uslov izvršeni slučajevi `True` i `False`. Opcija uključivanja se koristi za ograničavanje analize pokrivenosti na datoteke koje se nalaze unutar paketa aplikacije, što je jedini kod koji treba izmjeriti.

Bez opcije uključivanja, sve ekstenzije instalirane u virtuelnom okruženju i kod za same testove bili bi uključeni u izveštaje o pokrivenosti – a to bi dodalo mnogo buke izveštaju.

Nakon što su svi testovi izvršeni, funkcija `test()` piše izvještaj na konzolu i također piše ljepši HTML izvještaj na disk. HTML verzija prikazuje sav izvorni kod označen bojama koje označavaju linije koje su pokrivene testovima i one koje nisu.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti `git checkout 15a` da provjerite ovu verziju aplikacije. Da biste bili sigurni da imate instalirane sve zavisnosti, također pokrenite `pip install -r requirements/dev.txt`.

Slijedi primjer tekstu alnog izvještaja:

```
(venv) $ flask test --pokrivenost
...
-----
Obavljeni 23 testa za 6.337 sekundi

uredi
Sažetak pokrivenosti:
Ime           Stmts Miss Branch BrPart Cover
-----        -----
app/__init__.py      32      0      0      0 100%
api_v1/__init__.py   3       0      0      0 100%
authentication.py    29     18     10      0     28%
comments.py          40     30     12      0     19%
app/api_v1/decorators.py 11      3      2      0     62%
```

app/api_v1/errors.py app/	17	10	0	0	41%
api_v1/posts.py app/api_v1/	36	24	8	0	27%
users.py app/auth/__init__.py	30	24	12	0	14%
app/auth/forms.py app/auth/		0	0	0	100% 0
views.py app/decorators.py	3 45	8	8		70%
app/e-pošta .py app/	116	91	42	0	16%
exceptions.py app/main/	14	3	2	0	69%
__init__.py app/main/errors.py	15	9	0	0	40%
app/main/forms.py app/main/	2		0	0	100% 0
views.py app/models.py	6	0 1	0		83%
	20	15	6	0	19%
	39	7	6	0	71%
	178	140	34	0	18%
	236	42	42	6	79%
TOTAL	872	425	184	6	45%

HTML verzija: file:///home/flask/flasky/tmp/coverage/index.html

Izvještaj pokazuje ukupnu pokrivenost od 45%, što nije strašno, ali nije ni dobro. Klase modela, kojima je do sada posvećena sva pažnja testiranja jedinica, čine ukupno 236 izjava, od kojih je 79% pokriveno testovima. Očigledno, datoteke views.py u glavnom i auth nacrtu i rute u api_v1 nacrtu imaju vrlo nisku pokrivenost, budući da se ne koriste ni u jednom od postojećih jediničnih testova. I naravno, ove metrike pokrivenosti ne ukazuju na to koliko koda bez grešaka postoji u projektu, budući da drugi faktori (kao što je kvalitet testova) igraju veliku ulogu u tome.

Naoružani ovim izvještajem, lako je odrediti gdje testove treba dodati u test paket da bi se poboljšala pokrivenost—ali nažalost, ne mogu se svi dijelovi aplikacije testirati tako lako kao modeli baze podataka. Sljedeća dva odjeljka raspravljaju o naprednjim strategijama testiranja koje se mogu primjeniti na funkcije pregleda, obrasce i predloške.

Flask test klijent

Neki dijelovi koda aplikacije uvelike se oslanjaju na okruženje koje kreira pokrenuta aplikacija. Na primjer, ne možete jednostavno pozvati kod u funkciji prikaza da biste ga testirali, budući da funkcija možda treba da pristupi Flask kontekstualnim varijablama kao što su zahtjev ili sesija, možda očekuje podatke iz obrasca koji se nalaze u POST zahtjevu, a može također zahtijevaju prijavljenog korisnika. Ukratko, funkcije pregleda mogu se pokrenuti samo u kontekstu zahtjeva i pokrenute aplikacije.

Flask dolazi opremljen testnim klijentom koji pokušava riješiti ovaj problem, barem u određenoj mjeri. Test klijent replicira okruženje koje postoji kada se aplikacija izvodi unutar web servera, dozvoljavajući testovima da djeluju kao klijenti i šalju zahtjeve.

Funkcije prikaza ne vide nikakve velike razlike kada se izvršavaju pod test klijentom; zahtjevi se primaju i usmjeravaju na odgovarajuće funkcije pregleda, iz kojih

odgovori se generišu i vraćaju. Nakon što se funkcija pogleda izvrši, njen odgovor se prosljeđuje testu, koji može provjeriti je li ispravno.

Testiranje web aplikacija **Primjer**

15-2 pokazuje okvir za testiranje jedinica koji koristi test klijenta.

Primjer 15-2. tests/test_client.py: okvir za testove koji koriste Flask test klijent

```
import unittest
aplikacije import create_app, db
app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        odgovor = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertTrue('Stranger' in response.get_data(as_text=True))
```

U poređenju sa tests/test_basics.py, ovaj modul dodaje varijablu `self.client` instance, koja je Flask test klijentski objekat. Ovaj objekt izlaže metode koje izdaju zahtjeve aplikaciji. Kada je testni klijent kreiran sa omogućenom opcijom `use_cookies`, on će prihvatići i poslati kolačiće na isti način na koji to rade pretraživači, tako da se može koristiti funkcionalnost koja se oslanja na kolačiće za prisjećanje konteksta između zahtjeva. Konkretno, ovaj pristup omogućava korištenje korisničkih sesija koje su pohranjene u kolačićima.

`Test_home_page()` je jednostavan primjer onoga što testni klijent može učiniti. U ovom primjeru, izdaje se zahtjev za korijenski URL aplikacije. Povratna vrijednost metode `get()` testnog klijenta je objekt odgovora Flask koji sadrži odgovor koji je vratila pozvana funkcija pogleda. Da bi se provjerilo da li je test bio uspješan, provjerava se statusni kod odgovora, a zatim se u tijelu odgovora, dobivenom od `response.get_data()`, traži riječ "Stranger", koja je dio "Zdravo, Stranac!" pozdrav prikazan anonimnim korisnicima. Imajte na umu da `get_data()`

vraća tijelo odgovora kao niz bajtova po defaultu; passing `as_text=True` pretvara ga u string, sa kojim je lakše raditi.

Test klijent također može poslati POST zahtjeve koji uključuju podatke obrasca koristeći `post()` metodu, ali podnošenje obrazaca predstavlja malu komplikaciju. Kao što je objašnjeno u [Poglavlju 4](#), svi obrasci koje generira Flask-WTF imaju skriveno polje sa CSRF tokenom koje treba dostaviti zajedno sa formom. Da bi mogao poslati CSRF token, test bi trebao zatražiti stranicu koja prikazuje obrazac, zatim raščlaniti HTML vraćen u tom odgovoru i izdvajati token, kako bi ga zatim mogao poslati s podacima obrasca. Da biste izbjegli gnjavažu s bavljenjem CSRF tokenima u testovima, bolje je onemogućiti CSRF zaštitu u konfiguraciji testiranja. To je prikazano u [primjeru 15-3](#).

Primjer 15-3. `cong.py`: onemogućavanje CSRF zaštite u konfiguraciji testiranja

klasa `TestingConfig(Config): #...`

`WTF_CSRF_ENABLED` = Netačno

[Primer 15-4](#) prikazuje napredniji jedinični test koji simulira da novi korisnik registruje nalog, prijavljuje se, potvrđuje nalog tokenom za potvrdu i konačno se odjavljuje.

Primjer 15-4. `tests/test_client.py`: simulacija novog radnog toka korisnika sa Flask test klijentom

```
klasa FlaskClientTestCase(unittest.TestCase):
    ...
    # def test_register_and_login(self): #
        registrirajte odgovor na novi račun =
        self.client.post('/auth/register', data={
            'email': 'john@example.com',
            'korisničko ime': 'john', 'password':
            'mačka', 'password2': 'mačka'

    })
    self.assertEqual(response.status_code, 302)

    # prijavite se sa odgovorom novog
    # naloga = self.client.post('/auth/login', data={ 'email':
    #         'john@example.com', 'password': 'mačka' },
    #         follow_redirects=Tačno) self.assertEqual(response.status_code,
    # 200) self.assertTrue(re.search('Zdravo,\s+john!', response.get_data(as_text=True)))

    self.assertTrue( 'Još
        niste potvrdili svoj račun' u response.get_data(
        as_text=Tačno))
```

```

# pošalji token potvrde user =
User.query.filter_by(email='john@example.com').first().token =
user.generate_confirmation_token() response = self.client.get('/auth/confirm/
{} .format(token),
follow_redirects=True)
user.confirm(token)
self.assertEqual(response.status_code, 200)
self.assertTrue('Potvrdili ste svoj račun' u
response.get_data(as_text=True))

# log out
response = self.client.get('/auth/logout', follow_redirects=True)
self.assertEqual(response.status_code, 200) self.assertTrue('Odjavljeni ste' u
response.get_data(
as_text=True))

```

Test počinje slanjem obrasca na rutu registracije. Argument podataka za post() je rečnik sa poljima obrasca, koja moraju tačno odgovarati imenima polja definisanim u HTML obrascu. Budući da je CSRF zaštita sada onemogućena u konfiguraciji za testiranje, nema potrebe da šaljete CSRF token sa obrascem.

/auth/register ruta može odgovoriti na dva načina. Ako su podaci o registraciji ispravni, preusmjeravanje šalje korisnika na stranicu za prijavu. U slučaju nevažeće registracije, odgovor ponovo prikazuje stranicu sa obrascem za registraciju, uključujući sve odgovarajuće poruke o grešci. Da bi potvrdio da je registracija prihvaćena, test provjerava da li je statusni kod odgovora 302, što je kod za preusmjeravanje.

Drugi dio testa izdaje zahtjev za prijavu na aplikaciju koristeći e-poštu i lozinku koje ste upravo registrovali. Ovo se radi sa POST zahtjevom na /auth/login rutu. Ovaj put argument follow_redirects=True je uključen u post() poziv kako bi testni klijent radio kao pretraživač i automatski izdao GET zahtjev za preusmjereni URL. Sa ovom opcijom, statusni kod 302 neće biti vraćen; umjesto toga, vraća se odgovor sa preusmjerenog URL-a.

Uspješan odgovor na prijavu sada bi imao stranicu koja pozdravlja korisnika njegovim korisničkim imenom, a zatim ukazuje da je potrebno potvrditi račun da bi dobio pristup. Dvije izjave potvrđuju da je ovo vraćena stranica. Ovdje je zanimljivo primijetiti da se pretraga za stringom 'Halo, John!' ne bi funkcionalo jer je ovaj niz sastavljen od statičkih i dinamičkih delova, pa zbog načina na koji je Jinja2 šablon kreiran, konačni HTML ima dodatni razmak između ove dve reči. Da bi se izbjegla greška u ovom testu zbog razmaka, koristi se regularni izraz.

Sljedeći korak je potvrda računa, što predstavlja još jednu malu prepreku. URL za potvrdu se šalje korisniku putem e-pošte prilikom registracije, tako da ne postoji jednostavan način da mu se pristupi iz testa. Rješenje predstavljeno u testu zaobilazi token

koji je generiran kao dio registracije i generira još jedan direktno iz korisničke instance. Druga mogućnost bi bila da se token izdvoji raščlanjivanjem tijela e-pošte, koje Flask-Mail spremi kada radi u konfiguraciji za testiranje.

Sa tokenom pri ruci, sljedeći korak testa je simulacija korisnika koji klikne na URL tokena za potvrdu primljen putem e-pošte. Ovo se postiže slanjem GET zahtjeva na URL za potvrdu, koji uključuje token. Odgovor na ovaj zahtjev je preusmjeravanje na početnu stranicu, ali je još jednom navedeno follow_redirects=True , tako da testni klijent automatski traži preusmjerenu stranicu i vraća je. Odgovor se provjerava radi pozdrava i bljeskajuće poruke koja obavještava korisnika da je potvrda uspješna.

Posljednji korak u ovom testu je slanje GET zahtjeva na rutu za odjavu; da bi potvrdio da je ovo funkcionalo, test traži trepereću poruku u odgovoru.



Ako ste klonirali Git spremište aplikaciju na GitHub-u, možete pokrenuti git checkout 15b da provjerite ovu verziju aplikacije.

Testiranje web servisa

Flask test klijent se također može koristiti za testiranje RESTful web usluga. [Primjer 15-5](#) prikazuje primjer klase jediničnog testa sa dva testa.

[Primjer 15-5. tests/test_api.py: RESTful API testiranje sa Flask test klijentom](#)

```
klasa APITestCase(unittest.TestCase):
    ...
    # def get_api_headers(self, korisničko ime, lozinka):
    #     return {
    #         'Autorizacija': +
    #             'Osnovni ' b64encode(
    #                 (korisničko ime + ':' + lozinka).encode('utf-8')).decode('utf-8'), 'Prihvati':
    #                 'application/json', 'Content-Type': 'application/json'

    #     }

    def test_no_auth(self):
        odgovor = self.client.get(url_for('api.get_posts'),
                                   content_type='application/json')
        self.assertEqual(response.status_code, 401)

    def test_posts(self): #
        dodaj korisnika r =
            Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r)
```

```

u = Korisnik(email='john@example.com', lozinka='mačka', potvrđeno=Tačno,
uloga=r) db.session.add(u) db.session.commit()

# napišite
odgovor na post = self.client.post( '/
    api/v1/posts/',
    headers=self.get_api_headers('john@example.com', 'mačka'),
    data=json.dumps({'body ': 'tijelo *blog* posta'}))
self.assertEqual(response.status_code, 201) url =
response.headers.get('Location') self.assertIsNotNone(url)

# dobijte odgovor na
novi post = self.client.get( url,

    headers=self.get_api_headers('john@example.com', 'mačka'))
self.assertEqual(response.status_code, 200) json_response =
json.loads( response.get_data(as_text=True)) self.assertEqual('http://
localhost' + json_response['url'], url) self.assertEqual(json_response['body'],
'tijelo *blog* posta ') self.assertEqual(json_response['body_html'],
'<p>tijelo <em>blog</em> posta</p>')

```

Metode `setUp()` i `tearDown()` za testiranje API-ja su iste kao i za redovnu aplikaciju, ali podrška za kolačiće ne mora biti konfigurisana jer je API ne koristi. `Get_api_headers()` metoda je pomoćna metoda koja vraća uobičajena zaglavlja koja se moraju poslati s većinom API zahtjeva. To uključuje vjerodajnice za provjeru autentičnosti i zaglavlja vezana za MIME tip.

`Test_no_auth()` test je jednostavan test koji osigurava da je zahtjev koji ne uključuje vjerodajnice za autentifikaciju odbijen s kodom greške 401. `Test_posts()` test dodaje korisnika u bazu podataka, a zatim koristi RESTful API za umetanje objave na blogu i onda ga pročitaj nazad. Svi zahtjevi koji šalju podatke u tijelu moraju ih kodirati pomoću `json.dumps()`, jer Flask testni klijent ne kodira automatski u JSON.

Slično, tijela odgovora se također vraćaju u JSON formatu i moraju se dekodirati pomoću `json.loads()` prije nego što se mogu pregledati.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 15c da provjerite ovu verziju aplikacije.

End-to-End testiranje sa Selenom

Flask test klijent ne može u potpunosti emulirati okruženje pokrenute aplikacije.

Na primjer, bilo koja aplikacija koja se oslanja na JavaScript kod koji se izvodi u klijentskom pretraživaču neće raditi, jer će JavaScript kod uključen u odgovore biti vraćen testu bez izvršenja.

Kada testovi zahtevaju kompletno okruženje, nema drugog izbora osim da koristite pravi veb pretraživač povezan sa aplikacijom koja radi na pravom veb serveru. Na sreću, većina web pretraživača može biti automatizovana. [Selen](#) je alat za automatizaciju web pretraživača koji podržava najpopularnije web pretraživače u tri glavna operativna sistema. tems.

Python interfejs za Selen se instalira sa pip:

```
(venv) $ pip install selenium
```

Za Selenium je potreban upravljački program za željeni web pretraživač koji se instalira odvojeno, pored samog pretraživača. Postoje drajveri za sve glavne web pretraživače, tako da aplikacija može postaviti sofisticirani okvir za testiranje nekoliko pretraživača. Za ovu aplikaciju, međutim, samo će se web pretraživač Google Chrome koristiti za automatizovane testove, sa odgovarajućim drajverom, ChromeDriver. Ako koristite macOS računar sa instalaterom brew paketa, možete instalirati ChromeDriver na sljedeći način:

```
(venv) $ brew install chromedriver
```

Za Linux, Microsoft Windows ili macOS računar bez brew-a, možete preuzeti običan program za instalaciju ChromeDriver-a sa [ChromeDriver web stranice](#).

Testiranje sa Selenom zahtijeva da aplikacija radi unutar web servera koji osluškuje stvarne HTTP zahtjeve. Metoda koja će biti prikazana u ovom odeljku pokreće aplikaciju sa razvojnim serverom u pozadini dok se testovi izvode na glavnoj niti. Pod kontrolom testova, Selenium pokreće web pretraživač i povezuje ga sa aplikacijom kako bi izvršio potrebne operacije.

Problem s ovim pristupom je taj što nakon što su svi testovi završeni, Flask server treba zaustaviti, idealno na graciozan način, tako da pozadinski zadaci, kao što je mehanizam za pokrivanje koda, mogu čisto da završe svoj posao. Werkzeug web server ima opciju isključivanja, ali pošto server radi izolovan u sopstvenoj niti, jedini način da zatražite od servera da se isključi je slanje redovnog HTTP zahteva.

[Primjer 15-6](#) pokazuje implementaciju rute gašenja servera.

Primjer 15-6. _app/main/views.py: ruta gašenja servera

```
@main.route('/shutdown') def
server_shutdown(): ako nije
    current_app.testing: abort(404)
    shutdown =
    request.environ.get('werkzeug.server.shutdown') ako nije shutdown:
    abort(500) shutdown () return 'Isključivanje...'
```

Ruta za isključivanje će raditi samo kada aplikacija radi u režimu testiranja; pozivanje u drugim konfiguracijama će vratiti odgovor 404 statusnog koda. Stvarna procedura isključivanja uključuje pozivanje funkcije isključivanja koju Werkzeug izlaže u okruženju. Nakon pozivanja ove funkcije i povratka iz zahtjeva, razvojni web server će znati da mora elegantno izaći.

Primjer 15-7 pokazuje izgled testnog slučaja koji je konfiguiran za pokretanje testova sa Selebr.

Primjer 15-7. tests/test_selenium.py: okvir za testove koji koriste Selenium

```
iz webdrivera za uvoz selena

klasa SeleniumTestCase(unittest.TestCase): klijent =
    Nijedan

    @classmethod
    def setUpClass(cls): #
        start Chrome options
        = webdriver.ChromeOptions()
        options.add_argument('headless') try:
            cls.client =
                webdriver.Chrome(chrome_options=options)
        osim:
            pass

        # preskočite ove testove ako se pretraživač nije mogao pokrenuti ako
        cls.client: # kreirajte aplikaciju cls.app = create_app('testiranje')
        cls.app_context = cls.app.app_context() cls.app_context.push()

        # suzbija vođenje evidencije da bi jedinični rezultat bio čist.
        import logging logger = logging.getLogger('werkzeug')
        logger.setLevel("ERROR")
```

```

# kreirajte bazu podataka i popunite lažnim podacima db.create_all()

Role.insert_roles()
fake.users(10)
fake.posts(10)

# dodajte administratorskog korisnika
admin_role = Role.query.filter_by(permissions=0xff).first() admin =
Korisnik(email='john@example.com', korisničko ime='john', lozinka='mačka',
uloga=admin_role, potvrđeno=Tačno) db.session.add(admin)
db.session.commit()

# pokrenite Flask server u niti cls.server_thread
= threading.Thread( target=cls.app.run,
kwargs={'debug': 'false', 'use_reloader': False,
'use_debugger': False})

cls.server_thread.start()

@classmethod
def tearDownClass(cls): if
cls.client: # zaustavi
Flask server i pretraživač cls.client.get('http://
localhost:5000/shutdown') cls.client.quit() cls.server_thread.
pridruži se ()

# uništi bazu podataka
db.drop_all()
db.session.remove()

# ukloni kontekst aplikacije
cls.app_context.pop()

def setUp(self): ako
nije self.client:
self.skipTest('Web pretraživač nije dostupan')

def tearDown(self): pass

```

Metode klase setUpClass() i tearDownClass() se pozivaju prije i nakon izvršenja testova u ovoj klasi. Podešavanje uključuje pokretanje instance Chrome-a preko Selenium-ovog webdriver API-ja, i kreiranje aplikacije i baze podataka sa nekim početnim lažnim podacima za testiranje za korištenje. Aplikacija se pokreće u niti pomoću metode app.run(). Na kraju aplikacija prima zahtjev za /shutdown, što uzrokuje završetak pozadinske niti. Pretraživač se zatim zatvara i testna baza podataka se uklanja.



Pre nego što je uveden Flask interfejs komandne linije zasnovan na Click, morali ste da pokrenete Flask razvojni veb server pozivanjem app.run() iz glavne skripte aplikacije, ili u suprotnom koristite ekstenziju treće strane kao što je Flask. -Skripta. Dok je korištenje app.run() za pokretanje servera sada zamijenjeno naredbom flask run , metoda app.run() i dalje je podržana, a ovdje možete vidjeti kako još uvijek može biti korisna za složene situacije testiranja jedinica.



Selenium podržava mnoge druge web pretraživače osim Chrome-a. Pogledajte [dokumentaciju Selena](#) ako želite da koristite drugi web pretraživač ili testirate dodatne pretraživače.

Metoda setUp() koja se pokreće prije svakog testa preskače testove ako Selenium ne može pokrenuti web pretraživač u metodi startUpClass() . U [primjeru 15-8](#) možete vidjeti primjer testa napravljenog sa Selenom.

Primjer 15-8. tests/test_selenium.py: primjer jediničnog testa selena

```
klasa SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self): # id
        na početnu stranicu
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Zdravo,\s+Stranger!', self.client.page_source))

        # idite na stranicu za prijavu
        self.client.find_element_by_link_text('Prijava').click()
        self.assertIn('<h1>Prijava</h1>', self.client.page_source)

        # prijavite
        se self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('mačka')
        self.client.find_element_by_name('submit').click() self.assertTrue(re.search('Zdravo,
        \s+john!', self .client.page_source))

        # idite na stranicu profila korisnika
        self.client.find_element_by_link_text('Profil').click()
        self.assertIn('<h1>john</h1>', self.client.page_source)
```

Ovaj test se prijavljuje u aplikaciju koristeći administratorski nalog koji je kreiran u setUpClass() , a zatim otvara stranicu profila korisnika. Obratite pažnju na to koliko se metodologija testiranja razlikuje od Flask test klijenta. Prilikom testiranja sa Selenom, testovi šalju komande web pretraživaču i nikada ne stupaju u direktnu interakciju sa aplikacijom. The

komande se usko podudaraju sa radnjama koje bi pravi korisnik izvodio mišem ili tastaturom.

Test počinje pozivom get() sa početnom stranicom aplikacije. U pretraživaču ovo uzrokuje da se URL unese u adresnu traku. Da bi se potvrdio ovaj korak, izvor stranice se provjerava za "Zdravo, stranče!" pozdrav.

Da bi otisao na stranicu za prijavu, test traži vezu "Prijava" koristeći find_element_by_link_text(), a zatim poziva click() na njoj da bi pokrenuo pravi klik u pretraživaču. Selenium pruža nekoliko metoda pogodnosti find_element_by...() koje mogu pretraživati elemente unutar HTML stranice na različite načine.

Da bi se prijavio u aplikaciju, test locira polja obrasca e-pošte i lozinke po njihovim imenima koristeći find_element_by_name(), a zatim upisuje tekst u njih pomoću send_keys(). Obrazac se šalje pozivom click() na dugme za slanje. Personalizirani pozdrav se provjerava kako bi se osiguralo da je prijava bila uspješna i da je pretraživač sada na početnoj stranici.

Završni dio testa locira vezu „Profil“ u navigacijskoj traci i klikne je.

Da bi se potvrdilo da je stranica profila učitana, naslov s korisničkim imenom se pretražuje u izvoru stranice.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 15d da provjerite ovu verziju aplikacije. Ovo ažuriranje sadrži migraciju baze podataka, stoga ne zaboravite pokrenuti flask db upgrade nakon što provjerite kod. Da biste bili sigurni da imate instalirane sve zavisnosti, također pokrenite pip install -r requirements/dev.txt.

Kada pokrenete jedinične testove sa naredbom flask test , neće biti vidljive razlike. Jedinični test test_admin_home_page u [primjeru 15-8](#) će pokrenuti instancu Chromea bez glave i izvršiti sve radnje na njoj. Ako želite da vidite radnje koje se izvode u stvarnom Chrome prozoru, komentari sukladno takoj u [vezu](#) (commented-out) u tom podzoru.

Da li je vrijedno toga?

Do sada se možda pitate da li je testiranje pomoću Flask test klijenta ili Selena zaista vrijedno truda. To je ispravno pitanje i nema jednostavan odgovor.

Sviđalo vam se to ili ne, vaša aplikacija će biti testirana. Ako ga sami ne testirate, tada će vaši korisnici postati nevoljni testeri; oni će pronaći greške, a onda ćete ih morati popraviti pod pritiskom. Jednostavni i fokusirani testovi poput onih koji vježbaju modele baze podataka i druge dijelove aplikacije koji se mogu izvršiti van.

strane konteksta aplikacije uvijek treba provoditi, jer imaju vrlo nisku cijenu i osiguravaju pravilno funkcioniranje osnovnih dijelova logike aplikacije.

End-to-end testovi tipa koji Flask test klijent i Selenium mogu izvršiti ponekad su neophodni, ali zbog povećane složenosti njihovog pisanja, trebalo bi ih koristiti samo za funkcionalnost koja se ne može testirati izolovano. Aplikacioni kod treba da bude organizovan tako da je moguće ugorati poslovnu logiku u module aplikacije koji su nezavisni od konteksta aplikacije, te se na taj način mogu lakše testirati. Kod koji postoji u funkcijama pogleda trebao bi biti jednostavan i samo djelovati kao tanak sloj koji prihvaca zahtjeve i poziva odgovarajuće akcije u drugim klasama ili funkcijama koje inkapsuliraju logiku aplikacije.

Dakle, da, testiranje se apsolutno isplati. Ali važno je dizajnirati efikasnu strategiju testiranja i napisati kod koji to može iskoristiti.

Performanse

Niko ne voli spore aplikacije. Duga čekanja da se stranice učitaju frustriraju korisnike, pa je važno otkriti i ispraviti probleme u radu čim se pojave. U ovom poglavlju razmatraju se dva važna aspekta performansi web aplikacija.

Zapisivanje Spore performanse baze podataka

Kada performanse aplikacije polako degenerišu s vremenom, to je vjerovatno zbog sporih upita baze podataka, koji se pogoršavaju kako veličina baze podataka raste. Optimiziranje upita baze podataka može biti jednostavno kao dodavanje više indeksa ili složeno kao dodavanje keša između aplikacije i baze podataka. Izjava objašnjenja, dostupna u većini jezika za upite baze podataka, pokazuje korake koje baza podataka poduzima da izvrši dani upit, često otkrivajući neefikasnost u dizajnu baze podataka ili indeksa.

Ali prije nego počnete optimizirati upite, potrebno je odrediti koji su upiti oni koji vrijedi optimizirati. Tokom tipičnog zahteva može biti izdato nekoliko upita baze podataka, tako da je često teško identifikovati koji od svih upita su spori.

Flask-SQLAlchemy ima opciju za snimanje statistike o upitima baze podataka izdatim tokom zahtjeva. U [primjeru 16-1](#) možete vidjeti kako se ova funkcija može koristiti za evidentiranje upita koji su sporiji od konfigurisanog praga.

Primjer 16-1. app/main/views.py: prijavljivanje sporih upita baze podataka

```
iz flask_sqlalchemy import get_debug_queries

@main.after_app_request
def after_request(response):
    upit = get_debug_queries()
    ako query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
        current_app.logger.warning(
```

```
'Spor upit: %s\nParametri: %s\nTrajanje: %fs\nKontekst: %s\n' % (query.statement,
query.parameters, query.duration, query.context))
```

[povratni odgovor](#)

Ova funkcionalnost je pridružena rukovaocu `after_app_request`, koji radi na sličan način rukovatelju `before_app_request`, ali se poziva nakon što se vrati funkcija pogleda koja obrađuje zahtjev. Flask proslijedi objekt odgovora rukovatelju `after_app_request` u slučaju da ga treba modificirati.

U ovom slučaju, rukovatelj `after_app_request` ne mijenja odgovor; on samo dobija vremenske intervale upita koje je zabeležio Flask-SQLAlchemy i zatim beleži spore u loger aplikacije koji Flask postavlja na `app.logger`, pre nego što vrati odgovor, koji će zatim biti poslan klijentu.

Funkcija `get_debug_queries()` vraća upite izdate tokom zahtjeva kao listu. Informacije date za svaki upit prikazane su u [Tabeli 16-1](#).

Tabela 16-1. Statistika upita zabilježena od strane Flask-SQLAlchemy

Ime	Opis
izraz SQL izraz	
parametri	Parametri koji se koriste sa SQL izrazom
start_time	Vrijeme kada je upit izdat
end_time	Vrijeme kada je upit vratio duration
Trajanje	upita u sekundama
kontekstu	Niz koji označava lokaciju izvornog koda na kojoj je izdat upit

Obrađivač `after_app_request` hoda po listi i bilježi sve upite koji su trajali duže od praga danog u konfiguracijskoj varijabli `FLASKY_SLOW_DB_QUERY_TIME`.

Evidentiranje se izdaje na nivou upozorenja u ovoj aplikaciji, ali u nekim slučajevima može imati smisla tretirati spora upozorenja baze podataka kao greške.

Funkcija `get_debug_queries()` je podrazumevana omogućena samo u režimu za otklanjanje grešaka. Nažalost, problemi s performansama baze podataka rijetko se pojavljuju tokom razvoja jer se koriste mnogo manje baze podataka. Iz tog razloga je mnogo korisnije omogućiti ovu opciju u proizvodnji.

[Primjer 16-2](#) pokazuje promjene konfiguracije koje su potrebne da bi se omogućilo praćenje performansi upita baze podataka u proizvodnom načinu.

Primjer 16-2. cong.py: konfiguracija za sporo izvještavanje o upitima

```
konfiguracija klase :
# ...
SQLALCHEMY_RECORD_QUERIES = Tačno
FLASKY_SLOW_DB_QUERY_TIME = 0,5 #
...
```

SQLALCHEMY_RECORD_QUERIES govori Flask-SQLAlchemy da omogući snimanje statistike upita. Prag sporog upita postavljen je na pola sekunde. Obje konfiguracijske varijable su uključene u osnovnu klasu Config , tako da će biti omogućene za sve konfiguracije.

Kad god se otkrije spor upit, unos će biti upisan u Flask-ov aplikacijski loger. Da biste mogli pohraniti ove unose dnevnika, logger mora biti konfiguriran. Konfiguracija evidentiranja u velikoj mjeri ovisi o platformi na kojoj se nalazi aplikacija. Neki primjeri su prikazani u [Poglavlju 17.](#)



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 16a da provjerite ovu verziju aplikacije.

Izvorni kod Proling

Drugi mogući izvor problema s performansama je visoka potrošnja CPU-a, uzrokovan funkcijama koje izvode teške računare. Profileri izvornog koda korisni su u pronaalaženju najsporijih dijelova aplikacije. Profiler prati pokrenutu aplikaciju i bilježi funkcije koje se pozivaju i koliko dugo je svakoj potrebno da se pokrene. Zatim proizvodi detaljan izvještaj koji prikazuje najsporije funkcije.



Profiliranje se obično radi samo u razvojnem okruženju. Profiler izvornog koda čini da aplikacija radi mnogo sporije nego inače, jer mora da posmatra i beleži sve što se dešava u realnom vremenu. Profiliranje na proizvodnom sistemu se ne preporučuje, osim ako se ne koristi lagani profilator posebno dizajniran za rad u proizvodnom okruženju.

Flaskov razvojni web server, koji dolazi od Werkzeuga, može opcionalno omogućiti Python profiler za svaki zahtjev. [Primjer 16-3](#) dodaje novu opciju komandne linije u aplikaciju koja pokreće web server pod profilerom.

Primjer 16-3. flasky.py: pokretanje aplikacije pod priljubljenim zahtjevima

```
@app.cli.command()
@click.option('--length', default=25,
              help='Broj funkcija koje treba uključiti u izvještaj profilera.')
@click.option('--profile-dir', default=None,
              help='Direktorijum u koji se čuvaju fajlovi sa podacima profilera.')
def profile(length, profile_dir):
    """Pokreni
       aplikaciju pod profilerom koda.""" from werkzeug.contrib.profiler
    import ProfilerMiddleware app.wsgi_app =
    ProfilerMiddleware(app.wsgi_app, restrictions=[length], profile_dir=profile_dir)

    app.run(debug=False)
```

Ova naredba pričvršćuje ProfilerMiddleware iz Werkzeuga aplikaciji, preko svog atributa wsgi_app . WSGI međutim poziva se svaki put kada web server pošalje zahtjev aplikaciji i može modificirati način na koji se zahtjevom rukuje, u ovom slučaju hvatanjem podataka o profiliranju. Imajte na umu da se aplikacija tada programski pokreće pomoću metode app.run() .



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 16b da provjerite ovu verziju aplikacije.

Kada se aplikacija pokrene s flask profilom, konzola će prikazati statistiku profilera za svaki zahtjev, koji će uključivati 25 najsporijih funkcija. Opcija -length se može koristiti za promjenu broja funkcija prikazanih u izvještaju.

Ako je data opcija --profile-dir , podaci profila za svaki zahtjev se spremaju u datoteku u datom direktoriju. Datoteke podataka profilera mogu se koristiti za generiranje detaljnijih izvještaja koji uključuju graf poziva. Za više informacija o Python profilera, pogledajte [zvaničnu dokumentaciju](#).

Pripreme za raspoređivanje su završene. Sljedeće poglavje će vam dati pregled onoga što možete očekivati prilikom postavljanja vaše aplikacije.

Deployment

Server za web razvoj koji dolazi u paketu sa Flaskom nije dovoljno robustan, siguran ili efikasan za rad u proizvodnom okruženju. U ovom poglavlju se ispituju mogućnosti implementacije proizvodnje za Flask aplikacije.

Deployment Workow

Bez obzira na korišteni metod hostinga, postoji niz zadataka koji se moraju izvršiti kada se aplikacija instalira na proizvodni server. To uključuje kreiranje ili ažuriranje tabela baze podataka.

Ručno pokretanje ovih zadataka svaki put kada se aplikacija instalira ili nadograđi je sklonogreškama i oduzima mnogo vremena. Umjesto toga, komanda koja izvršava sve potrebne zadatke može se dodati flasky.py.

Primjer 17-1 pokazuje implementaciju naredbe deploy koja je prikladna za Flasky.

Primjer 17-1. flasky.py: naredba postavljanja

```
from flask_migrate import upgrade
from app.models import Role, User

@manager.command
def deploy():
    """Pokreni zadatke postavljanja."""
    # migrira bazu podataka na nadogradnju
    # najnovije revizije ()
    # kreirajte ili ažurirajte korisničke
    # uloge Role.insert_roles()
```

```
# osigurajte da svi korisnici sami sebe prate
User.add_self_follows()
```

Sve funkcije koje poziva ova naredba su kreirane prije; oni se samo pozivaju zajedno iz jedne naredbe kako bi se pojednostavila implementacija aplikacije.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17a da provjerite ovu verziju aplikacije.

Sve ove funkcije su dizajnirane na način da ne uzrokuju probleme ako se izvrše više puta. Dizajniranje funkcija ažuriranja na ovaj način omogućava pokretanje samo ove naredbe deploy svaki put kada se izvrši instalacija ili nadogradnja bez potrebe da brinete o nuspojavama uzrokovanim funkcijom koja se pokreće u pogrešno vrijeme.

Evidentiranje grešaka tokom proizvodnje

Kada aplikacija radi u modu za otklanjanje grešaka, Werkzeugov interaktivni program za otklanjanje grešaka pojavljuje se kad god dođe do greške. Trag steka greške se prikazuje na web stranici, a moguće je pogledati izvorni kod, pa čak i procijeniti izraze u kontekstu svakog okvira steka koristeći Flask interaktivni web-bazirani debager.

Debugger je odličan alat za otklanjanje grešaka u aplikaciji tokom razvoja, ali očigledno se ne može koristiti u proizvodnji. Greške koje se javljaju u proizvodnji se prešućuju u umesto toga korisnik dobija diskretnu stranicu greške kod 500. Ali srećom, tragovi steka ovih grešaka nisu potpuno izgubljeni, pošto ih Flask upisuje u log fajl.

Tokom pokretanja, Flask kreira instancu Pythonove klase `logging.Logger` i prilaže je instanci aplikacije kao `app.logger`. U modu za otklanjanje grešaka, ovaj logger piše u konzolu, ali u proizvodnom modu nema rukovatelja koji su za njega konfigurirani po defaultu. Osim ako nije dodan rukovalac, dnevnići se ne pohranjuju. Promjene u [primjeru 17-2](#) konfigurišu rukovalac evidentiranjem koji šalje greške koje se javljaju tokom izvođenja pod proizvodnom konfiguracijom na adresu e-pošte administratora konfigurisanu u postavci `FLASKY_ADMIN`.

Primjer 17-2. cong.py: slanje e-pošte za greške u aplikaciji

```
klasa ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
        Config.init_app(app)
```

```

# greška e-pošte administratorima import
logging from logging.handlers import
SMTPHandler vjerodajnice = Nijedan siguran =
Nema ako getattr(cls, 'MAIL_USERNAME', None)
nije Ništa:

vjerodajnice = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD) ako
getattr(cls, 'MAIL_USE_TLS', Ništa): siguran = () mail_handler =
SMTPHandler()

mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT ),
fromaddr=cls.FLASKY_MAIL_SENDER,
toaddrs=[cls.FLASKY_ADMIN],
subject=cls.FLASKY_MAIL_SUBJECT_PREFIX +
vjerodajnice=akreditivi, securemail_levelure .
logger.addHandler(mail_handler)

```

Podsjetimo da sve konfiguracijske klase imaju statičku metodu init_app() koju poziva create_app(), koja do sada nije korištena. U implementaciji ove metode za klasu ProductionConfig , dnevnik aplikacije je sada konfiguriran s rukovaocem dnevnika koji šalje greške primaocu e-pošte.

Nivo evidentiranja e-mail logera je postavljen na logging.ERROR, tako da će se samo ozbiljni problemi slati e-poštom. Poruke evidentirane na nižim nivoima mogu se evidentirati u datoteku, sistemski dnevnik ili bilo koje drugo podržano odredište dodavanjem odgovarajućih rukovatelja evidentiranjem. Metoda evidentiranja koja se koristi za ove poruke u velikoj mjeri ovisi o hosting platformi.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17b da provjerite ovu verziju aplikacije.

Cloud Deployment

Trend u hostingu aplikacija je hostovanje „u oblaku“, ali to može značiti mnogo različitih stvari. Na najosnovnijem nivou, cloud hosting može značiti da je aplikacija instalirana na jednom ili više virtualnih servera, koji za sve namjere i svrhe rade i osjećaju se kao fizičke mašine, ali u stvarnosti su virtualne mašine kojima upravlja cloud operater. Primjer ovih tipova servera su oni koji su dostupni putem usluge EC2 od Amazon Web Services (AWS). Postavljanje aplikacije na virtualni server slično je tradicionalnom postavljanju na namenski server, kao što je opisano kasnije u ovom poglavljju.

Napredniji model implementacije baziran je na kontejnerima. Kontejner izoluje aplikaciju u sliku aplikacije i njenog okruženja. Slika kontejnera uključuje aplikaciju plus sve zavisnosti koje su joj potrebne za pokretanje. Kontejnerska platforma, kao što je Docker, tada može instalirati i izvršiti unaprijed generiranu sliku kontejnera na bilo kojem sistemu u kojem se pokreće.

Druga opcija implementacije, formalno poznata kao Platforma kao usluga (PaaS), oslobađa programere aplikacija od svakodnevnih zadataka instaliranja i održavanja hardverskih i softverskih platformi na kojima aplikacija radi. U PaaS modelu, provajder usluga nudi potpuno upravljanu platformu na kojoj aplikacije mogu da rade. Sve što programer aplikacije treba da uradi je da otpremi kod aplikacije na servere koje održava provajder, nakon čega on automatski postaje dostupan, obično u roku od nekoliko sekundi. Većina PaaS provajdera nudi načine za dinamičko „skalairanje“ aplikacije dodavanjem ili uklanjanjem servera po potrebi kako bi se održao korak s brojem primljenih zahtjeva.

Ostatak ovog poglavlja nudi uvod u Heroku (jedan od najpopularnijih PaaS provajdera), Docker kontejnere i na kraju tradicionalne implementacije, koje su pogodne za namenske ili virtuelne servere.

Heroku platforma

Heroku je bio jedan od prvih PaaS provajdera, koji posluje od 2007. Heroku platforma je veoma fleksibilna i podržava dugu listu programskih jezika, uključujući Python. Da bi postavio aplikaciju na Heroku, programer koristi Git da gurne aplikaciju na Herokuov specijalni Git server, koji automatski pokreće instalaciju, nadogradnju, konfiguraciju i implementaciju aplikacije.

Heroku koristi računarske jedinice zvane dynos za mjerjenje korištenja i naplatu usluge. Najčešći tip dyno-a je web dyno, koji predstavlja instancu web servera. Aplikacija može povećati svoj kapacitet obrade zahtjeva postavljanjem više web dynosa, od kojih svaki pokreće instancu aplikacije. Druga vrsta dyno-a je radni dyno uređaj, koji se koristi za obavljanje pozadinskih poslova ili drugih zadataka podrške.

Platforma pruža veliki broj dodataka i dodataka za baze podataka, podršku putem e-pošte i mnoge druge usluge. Sljedeći odjeljci proširuju neke od detalja uključenih u postavljanje Flaskyja na Heroku.

Priprema aplikacije Za rad sa

Herokuom, aplikacija mora biti smještena u Git spremištu. Ako radite sa aplikacijom koja je hostovana na udaljenom Git serveru, kao što je GitHub ili Bitbucket, kloniranje aplikacije će kreirati lokalno Git spremište koje je savršeno za

koristiti sa Herokuom. Ako aplikacija već nije hostovana u Git repozitorijumu, moraćete da kreirate jednu za nju na vašoj mašini za razvoj.



Ako planirate da hostujete svoju aplikaciju na Heroku-u, dobra je ideja da počnete da koristite Git od samog početka. GitHub ima vodič za instalaciju i podešavanje za tri glavna operativna sistema u svom **vodiču za pomoć**.

Kreiranje Heroku naloga

Morate **kreirati nalog sa Herokuom** prije nego što budete mogli koristiti uslugu. Heroku pruža besplatan nivo koji vam omogućava da ugostite nekoliko jednostavnih aplikacija, tako da je ovo odlična platforma za eksperimentisanje.

Instaliranje Heroku CLI

Za rad sa Heroku uslugom, **Heroku CLI** mora biti instaliran. Ovo je klijent komandne linije koji upravlja interakcijama sa uslugom. Heroku obezbeđuje instalatere za tri glavna operativna sistema.

Prva stvar koju treba uraditi nakon instaliranja CLI je da se autentifikujete sa svojim Heroku nalogom putem heroku login komande:

```
$ heroku login  
Unesite svoje Heroku akreditive.  
E-mail: <vaša-e-adresa> Lozinka:  
<vaša-lozinka>
```



Važno je da vaš SSH javni ključ bude učitan u Heroku, jer je to ono što omogućava git push komandu. Obično naredba za prijavu automatski kreira i otprema SSH javni ključ, ali naredba heroku keys:add može se koristiti za učitavanje vašeg javnog ključa odvojeno od naredbe za prijavu ili ako trebate učitati dodatne ključeve.

Kreiranje aplikacije

Sljedeći korak je kreiranje aplikacije. Prije nego što se to učini, aplikacija mora biti pod Git izvornom kontrolom. Ako ste koristili GitHub spremište za praćenje koda u ovoj knjizi, onda već imate Git spremište. Ako ne, morat ćete ga kreirati sada. Da biste registrovali aplikaciju kod Herokua, pokrenite sljedeću naredbu iz direktorija najvišeg nivoa aplikacije:

```
$ heroku create <appname>  
Kreiranje <appname>... urađeno  
https://<appname>.herokuapp.com/ | https://git.heroku.com/<ime aplikacije>.git
```

Nazivi Heroku aplikacija moraju biti jedinstveni za sve korisnike, tako da morate smisliti ime koje ne preuzima nijedna druga aplikacija. Kao što je naznačeno u izlazu naredbe `create`, nakon implementacije aplikacija će biti dostupna na <https://<app-name>.herokuapp.com>. Heroku također podržava korištenje prilagođenog imena domene za vašu aplikaciju.

Kao dio kreiranja aplikacije, Heroku kreira Git server posvećen vašoj aplikaciji na <https://git.heroku.com/<appname>.git>. Naredba `create` dodaje ovaj server u vaše lokalno Git spremište kao git daljinski s imenom `heroku`:

```
$ git daljinski show heroku *
daljinski heroku
Dohvati URL: https://git.heroku.com/<appname>.git Push
URL: https://git.heroku.com/<appname>.git GLAVA grana:
(nepoznato)
```

Komanda `flask` zahtijeva da varijabla okruženja `FLASK_APP` bude postavljena da radi. Da biste bili sigurni da su sve naredbe koje se izvršavaju u Heroku okruženju uspješne, dobra je ideja registrirati ovu varijablu okruženja tako da je uvijek postavljena kada Heroku izvršava komande povezane s ovom aplikacijom. Ovo se može uraditi naredbom config :

```
$ heroku config:set FLASK_APP=flasky.py
Postavljanje FLASK_APP i ponovno pokretanje <appname>...
urađeno, v4 FLASK_APP: flasky.py
```

Omogućavanje baze

podataka Heroku podržava Postgres baze podataka kao dodatak. Nivo besplatne usluge uključuje malu bazu podataka do 10.000 redova. Da priložite Postgres bazu podataka vašoj aplikaciji, koristite sljedeću naredbu:

```
$ heroku addons:create heroku-postgresql:hobby-dev Kreiranje
heroku-postgresql:hobby-dev na <appname>... besplatna baza podataka
je kreirana i dostupna
! Ova baza podataka je prazna. Ako nadogradite, možete prenijeti!
  podaci iz druge baze podataka sa pg:copy Kreiran postgresql-
  cubic-41298 kao DATABASE_URL Koristite heroku addons:docs heroku-
  postgresql za pregled dokumentacije
```

Kao što pokazuje izlaz naredbe, kada se aplikacija pokrene unutar Heroku platforme, vidjet će lokaciju baze podataka i vjerodajnice u varijabli okruženja `DATABASE_URL`. Format ove varijable je URL, tačno u formatu koji SQLAlchemy očekuje. Podsjetimo da `cong.py` skripta koristi vrijednost `DATABASE_URL` ako je definirana, tako da će veza sa Postgres bazom podataka raditi automatski.

Podešavanje

evidencije Evidentiranje fatalnih grešaka putem e-pošte dodato je ranije, ali je pored toga važno konfigurisati evidenciju manjih kategorija poruka. Dobar primjer ovih vrsta poruka su upozorenja za spore upite baze podataka dodana u [poglavlju 16.](#)

Heroku uzima u obzir svaki izlaz koji je aplikacija napisala u stdout ili stderr dnevниke, tako da je potrebno dodati rukovalac evidencije da bi se generirao ovaj izlaz. Heroku snima izlaz evidencije i postaje dostupan preko Heroku klijenta pomoću naredbe heroku logs .

Konfiguracija evidentiranja može se dodati u klasu ProductionConfig u njenoj statičkoj metodi init_app() . Ali pošto je ova vrsta evidentiranja specifična za Heroku, bolji pristup je da se definiše nova konfiguracija posebno za ovu platformu, ostavljajući ProductionConfig kao osnovnu konfiguraciju za različite tipove proizvodnih platformi. Klasa HerokuConfig je prikazana u [primjeru 17-3.](#)

Primjer 17-3. cong.py: Heroku konguracija

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)
```

Kada Heroku izvrši aplikaciju, mora znati da se ova nova konfiguracija mora koristiti. Instanca aplikacije kreirana u flasky.py koristi varijablu okruženja FLASK_CONFIG da zna koju konfiguraciju treba koristiti, tako da ova varijabla mora biti postavljena na odgovarajući način u Heroku okruženju. Varijable okruženja za Heroku okruženje se postavljaju pomoću Heroku klijentove naredbe config:set :

```
$ heroku config:set FLASK_CONFIG=heroku
Postavljanje FLASK_CONFIG i ponovno pokretanje <appname>... gotovo,
v4 FLASK_CONFIG: heroku
```

Da biste povećali sigurnost vaše aplikacije, dobra je ideja konfigurirati niz koji je teško pogoditi kao tajni ključ aplikacije, koji se koristi za potpisivanje korisničke sesije i tokena za autentifikaciju. Osnovna klasa Config uključuje atribut SECRET_KEY za ovu svrhu i postavlja svoju vrijednost iz varijable okruženja istog imena ako postoji. Kada radite na aplikaciji u vašem razvojnog sistemu, u redu je ostaviti ovu varijablu nedefiniranu i pustiti klasi Config da konfigurira tvrdo kodiran

vrijednost, ali na proizvodnjoj platformi izuzetno je važno postaviti jak tajni ključ koji nikome nije poznat, jer će procureti ključ omogućiti napadaču da krije se u pozadini i koristi se sesije ili generira važeće tokene. Da bi vaš ključ bio siguran, samo postavite varijablu okruženja SECRET_KEY na jedinstveni niz koji nije nigdje pohranjen:

```
$ heroku config:set SECRET_KEY=d68653675379485599f7876a3b469a57
Postavljanje SECRET_KEY i ponovno pokretanje <appname>... urađeno, v4
SECRET_KEY: d68653675379485599f786
```

Postoji mnogo načina za generiranje nasumičnih nizova koji su prikladni da se koriste kao tajni ključevi. To možete učiniti sa Python-om na sljedeći način:

```
(venv) $ python -c "uvoz uuid; print(uuid.uuid4().hex)"
d68653675379485599f7876a3b469a57
```

Konfigurisanje e-

pošte Heroku ne obezbeđuje SMTP server, tako da se mora konfigurisati eksterni server. Postoji nekoliko dodataka trećih strana koji integriraju podršku za slanje e-pošte spremnu za proizvodnju sa Herokuom, ali za potrebe testiranja i evaluacije dovoljno je koristiti zadani Gmail konfiguraciju naslijedenu iz osnovne Config klase.

Budući da može predstavljati sigurnosni rizik ugrađivanje akreditiva za prijavu direktno u skriptu, korisničko ime i lozinka za pristup Gmail SMTP serveru su dati kao varijable okruženja (ako još niste, vrlo je dobra ideja da umjesto koristeći svoj lični nalog e-pošte kreirate sekundarnu e-poštu koju ćete koristiti za testiranje):

```
$ heroku config:set MAIL_USERNAME=<vaše-gmail-korisničko ime> $
heroku config:set MAIL_PASSWORD=<vaša-gmail-lozinka>
```

Dodavanje zahteva najvišeg nivoa le

Heroku instalira zavisnosti paketa iz datoteke requirements.txt pohranjene u direktorijumu najvišeg nivoa aplikacije. Sve zavisnosti u ovoj datoteci će biti uvezene u virtuelno okruženje kojim upravlja Heroku kao deo implementacije.

Heroku datoteka sa zahtjevima mora uključivati sve uobičajene zahtjeve za proizvodnju verziju aplikacije, plus psycopg2 paket koji omogućava SQL-Alchemy pristup Postgres bazi podataka. Datoteka heroku.txt sa ovim ovisnostima može se dodati u direktorij zahtjeva i zatim uvesti iz datoteke requirements.txt najvišeg nivoa kao što je prikazano u [primjeru 17-4](#).

Primjer 17-4. Zahtevi.txt: Heroku zahtjevi le

```
-r zahtjevi/heroku.txt
```

Omogućavanje bezbednog HTTP-

a uz Flask-SSLify Kada se korisnik prijavi u aplikaciju slanjem korisničkog imena i lozinke u web obrascu, ove vrednosti su u opasnosti da ih presretne zlonamerna treća strana, kao što je već nekoliko puta objašnjeno. Tokom razvoja to nije problem, ali ovaj rizik treba eliminisati kada implementirate aplikaciju na proizvodni server. Da biste spriječili otkrivanje korisničkih vjerodajnica dok su u tranzitu, potrebno je koristiti siguran HTTP, koji šifrira svu komunikaciju između klijenata i servera koristeći kriptografiju javnog ključa.

Heroku čini sve aplikacije kojima se pristupa na domeni herokuapp.com dostupnim na http:// i https:// bez potrebe za konfiguracijom. Budući da aplikacija radi na Heroku domeni, koristit će Herokuov vlastiti SSL certifikat. Jedina neophodna radnja za potpunu sigurnost aplikacije je presretanje svih zahtjeva poslatih na http:// sučelje i preusmjeravanje na https://, što je upravo ono što Flask SSLify ekstenzija radi.

Kao i obično, Flask-SSLify se instalira sa pip:

```
(venv) $ pip install flask-sslify
```

Kod koji aktivira ovu ekstenziju se dodaje funkciji tvornice aplikacije, kao što je prikazano u [primjeru 17-5](#).

[Primjer 17-5. app/__init__.py: preusmjeravanje svih zahtjeva na sigurni HTTP](#)

```
def create_app(config_name):
    # ...
    ako app.config['SSL_REDIRECT']:
        from flask_sslify import SSLify
        SSLify(app)
    # ...
```

Podršku za SSL treba omogućiti samo u produksijskom modu i samo kada ga platforma podržava. Kako bi se olakšalo uključivanje i isključivanje SSL-a, dodaje se nova konfiguracijska varijabla pod nazivom SSL_REDIRECT. Osnovna klasa Config postavlja je na False, tako da se SSL preusmjeravanja ne koriste po defaultu, a klasa HerokuConfig je nadjačava tako da se samo na toj konfiguraciji izdaju preusmjeravanja. Implementacija ove varijable konfiguracije prikazana je u [primjeru 17-6](#).

Primjer 17-6. cong.py: podešavanje upotrebe SSL-a

```
konfiguracija klase :
# ...
SSL_REDIRECT = Netačno

klasa HerokuConfig(ProductionConfig):
# ...
SSL_REDIRECT = Tačno ako os.environ.get('DYNO') drugo Netačno
```

Vrijednost SSL_REDIRECT u HerokuConfig je postavljena na True samo ako postoji varijabla okruženja DYNO . Ovu varijablu postavlja Heroku u svom okruženju, tako da korištenje Heroku konfiguracije za lokalno testiranje ne aktivira SSL preusmjeravanja.

Sa ovim promjenama, korisnici će biti primorani da koriste SSL server kada pristupaju aplikaciji na Heroku-u—ali postoji još jedan detalj koji treba da se obradi kako bi ova podrška bila potpuna. Kada koristite Heroku, klijenti se ne povezuju direktno na aplikaciju, već na reverzni proxy server. Obrnuti proxy server prima zahtjeve od mnogih aplikacija i prosljeđuje ih svakoj od njih prema potrebi. U ovoj vrsti podešavanja, samo proxy server radi u SSL modu; SSL veza se prekida na proxy serveru, a aplikacije primaju prosljeđene zahtjeve od proxy servera bez enkripcije. Ovo predstavlja problem kada aplikacija treba da generiše apsolutne URL-ove, jer u Flask aplikaciji objekat zahteva opisuje prosleđeni zahtev, koji nije šifrovan, a ne originalni zahtev koji klijent šalje preko šifrovane veze.

Primjer problema koji ovo može uzrokovati je generiranje veza za potvrdu naloga ili za poništavanje lozinke koje se šalju e-poštom korisnicima. Kada se url_for() pozove sa _external=True da generiše apsolutni URL za ove veze, Flask će koristiti http:// za njih, jer ne zna da postoji obrnuti proxy koji prihvata šifrovane veze izvana .

Proxy serveri prosljeđuju informacije koje opisuju originalni zahtjev od klijenta do preusmjerenih web servera kroz prilagođena HTTP zaglavљa, tako da je moguće utvrditi da li korisnik komunicira s aplikacijom preko SSL-a gledajući ova zaglavљa. Werkzeug pruža WSGI međuverski softver koji provjerava prilagođena zaglavљa sa proxy servera i u skladu s tim ažurira objekt zahtjeva tako da, na primjer, request.is_secure odražava stanje šifriranja zahtjeva koji je klijent poslao obrnutom proxy serveru i ne zahtjev koji je proxy server zatim prosljedio aplikaciji. **Primjer 17-7** pokazuje kako dodati ProxyFix srednji softver u aplikaciju.

Primjer 17-7. cong.py: dodavanje podrške za proxy servere

```
klasa HerokuConfig(ProductionConfig):
    ...
    # @classmethod
    def init_app(cls, app):
        ...

        # obradi zaglavja obrnutog proxy servera iz
        werkzeug.contrib.fixers import ProxyFix app.wsgi_app
        = ProxyFix(app.wsgi_app)
```

Međuover se dodaje u metodu inicijalizacije za Heroku konfiguraciju.

WSGI međuvera kao što je ProxyFix se dodaje omotavanjem WSGI aplikacije.

Kada dođe zahtjev, međuoprema ima priliku da pregleda okruženje i izvrši promjene prije nego što se zahtjev obradi. ProxyFix Middleware je neophodan ne samo za Heroku, već i za svaku implementaciju koja koristi obrnuti proxy server.

Pokretanje proizvodnog web

servera Heroku očekuje od aplikacija da pokrenu vlastiti proizvodni web server i konfigurišu ga da sluša zahtjeve na broju porta postavljenom u varijabli okruženja PORT.

Razvojni web server koji dolazi sa Flaskom će se u ovoj situaciji ponašati vrlo loše jer nije dizajniran za rad u proizvodnom okruženju. Dva web servera spremna za proizvodnju koji dobro rade sa Flask aplikacijama su [Gunicorn](#) i [uWSGI](#).

Dobro je instalirati odabrani web server u lokalnom virtuelnom okruženju, kako bi se mogao testirati na sličan način kao što će raditi u Heroku okruženju.

Na primjer, Gunicorn se instalira na sljedeći način:

```
(venv) $ pip install gunicorn
```

Da biste pokrenuli aplikaciju lokalno pod Gunicorn, koristite sljedeću naredbu:

```
(venv) $ gunicorn flasky:app
[2017-08-03 23:54:36 -0700] [INFO] Pokretanje gunicorn 19.7.1 [2017-08-03
23:54:36 -0700] [INFO] Slušanje na: http://127.0.0.1:8000 (68982)
[2017-08-03 23:54:36 -0700] [INFO] Korišćenje radnika:
sinhronizacija [2017-08-03 23:54:36 -0700] [INFO] Pokretanje radnika sa pid-om: 68985
```

Argument flasky:app govori Gunicorn-u gdje se nalazi instanca aplikacije.

Ime dato prije dvotočke je paket ili modul koji definira ovu instancu, dok je ime nakon dvotočke stvarno ime instance aplikacije. Imajte na umu da Gunicorn podrazumevano koristi port 8000, a ne 5000 kao Flask. Kao i web server za razvoj Flask, možete izaći iz Gunicorn-a pomoću Ctrl+C.



Gunicorn web server ne radi na Microsoft Windows-u.
 Drugi preporučeni web server, uWSGI, radi na Windowsima, ali može biti teško instalirati jer je napisan u izvornom kodu. Ako želite da testirate Heroku implementaciju na vašem Windows sistemu, možete koristiti [Waitress](#), koji je još jedan čisti Python web server koji je na mnogo načina sličan Gunicorn-u, ali ima prednost što u potpunosti podržava Windows. Konobarica je instalirana sa pip:

```
(venv) $ pip install konobarica
```

Da biste pokrenuli Web server Waitress, koristite naredbu konobarica-serve :

```
(venv) $ waitress -serve --port 8000 flasky:app
```

Dodavanje

Procle Heroku mora znati koju naredbu koristiti za pokretanje aplikacije. Ova komanda je data u posebnoj datoteci pod nazivom Procle. Ova datoteka mora biti uključena u direktorij naviše razine aplikacije.

Primjer 17-8 prikazuje sadržaj ove datoteke.

Primjer 17-8. Procle: Heroku Procle

```
web: gunicorn flasky:app
```

Format za Procle je vrlo jednostavan: u svakoj liniji se daje ime zadatka, nakon čega slijedi dvotočka, a zatim naredba koja pokreće zadatak. Naziv zadatka web je poseban; Heroku ga prepoznaće kao zadatak koji pokreće web server. Heroku će ovom zadatku dati varijablu okruženja PORT postavljenu na port na kojem aplikacija treba da sluša zahtjeve. Gunicorn po defaultu poštjuje varijablu PORT ako je postavljena u okruženju, tako da nema potrebe da je uključujete u naredbu za pokretanje.



Ako koristite Microsoft Windows ili trebate da vaša aplikacija bude u potpunosti kompatibilna s tom platformom, umjesto toga možete koristiti [Waitress](#) web server:

```
web: waitress-serve --port=$PORT flasky:app
```



Aplikacije mogu deklarirati dodatne zadatke s drugim nazivima osim web u Procleyu. Svaki zadatak uključen u Procleyu će biti pokrenut na vlastitom dyno-u.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17c da provjerite ovu verziju aplikacije. Ako koristite Microsoft Windows, pokrenite git checkout 17c waitress da provjerite verziju aplikacije koja je konfiguirana da koristi Waitress web server umjesto Gunicorn-a.

Testiranje pomoću Heroku Local

-a Heroku CLI uključuje lokalnu komandu, koja se koristi za lokalno pokretanje aplikacije na vrlo sličan način na koji radi na Heroku serverima. Međutim, varijable okruženja kao što je FLASK_APP nisu dostupne kada se aplikacija pokreće lokalno. Lokalna komanda heroku traži varijable okruženja koje konfigurišu aplikaciju u datoteci pod nazivom .env u direktorijumu najvišeg nivoa aplikacije. Na primjer, .env datoteka može sadržavati sljedeće varijable:

```
FLASK_APP=flasky.py
FLASK_CONFIG=heroku
MAIL_USERNAME=<vaše-gmail-korisničko ime>
MAIL_PASSWORD=<vaša-gmail-lozinka>
```



Budući da .env datoteka sadrži lozinke i druge osjetljive informacije o računu, nikada je ne bi trebalo dodavati u kontrolu izvora.

Prije nego što se aplikacija može pokrenuti, potrebno je izvršiti zadatak postavljanja za postavljanje baze podataka. Jednokratni zadaci se mogu izvršiti naredbom local:run :

```
(venv) $ heroku local:run flask deploy [OKAY] Učitana
ENV .env datoteka kao KEY=VALUE Format INFO Kontekst impl
SQLiteImpl.
INFO Pretpostavlja netransakpcioni DDL.
Info Pokretanje -> 38c4e85512A9, Početne informacije o migraciji 38C4E85512A9 ->
456A945560F6, Podrška za prijavu 456A945560111 -> 1901636271 -> 56ed7d33de8d, korisničke
uloge 56D7D33DE8D -> D66F086B258, Informacije o korisniku, Informacije o korisniku Trčanje nadogradnje
D66F086B258 -> 198b0eebcf9, avatarske hase info trčanje na nadogradnji 198B0EEBCF9 -> 1b966e7f4b9e ->
288cd3dc5a8, bogat dograditi 288cd3dc5a8ea, sljedbeni info rada 2356A38169EA -> 51F5CCFB190, komentari
```

Lokalna komanda heroku čita Procfile i izvršava zadatke definirane njime:

```
(venv) $ heroku localni
[OKAY] Učitana ENV .env datoteka kao KEY=VRIJEDNOST
Format 11:37:49 AM web.1 | [INFO] Pokretanje gunicorn 19.7.1
11:37:49 web.1 | [INFO] Slušam na: http://0.0.0.0:5000 (91686)
11:37:49 AM web.1 | [INFO] Korišćenje radnika:
sinhronizacija 11:37:49 AM web.1 | [INFO] Booting worker sa pid-om: 91689
```

Izlaz evidencije svih zadataka pokrenutih ovom naredbom se konsoliduje u jedan tok koji se ispisuje na konzolu, sa svakim redom sa prefiksom vremenske oznake i naziva zadatka.

Lokalna komanda heroku također omogućava simulaciju upotrebe višestrukih dynosa za skaliranje aplikacije. Sljedeća naredba pokreće tri web radnika, od kojih svaki sluša na drugom portu:

```
(venv) $ heroku localni web=3
```

Implementacija pomoću git

push -a Poslednji korak u procesu je učitavanje aplikacije na Heroku servere. Uvjerite se da su sve promjene predane u lokalno Git spremište, a zatim koristite git push heroku master da prenesete aplikaciju na heroku daljinski:

```
$ git push heroku master
Brojanje objekata: 502, gotovo.
Delta kompresija pomoću do 8 niti.
Kompresija objekata: 100% (426/426), urađeno.
Predmeti za pisanje: 100% (502/502), 108.03 KiB | 0 bajtova/s, gotovo.
Ukupno 502 (delta 303), ponovo korišćeno 146 (delta 61)
daljinsko: Kompresovanje izvornih fajlova... završeno.
daljinski: Izvor zgrade: daljinski

daljinski: -----> Python aplikacija otkrivena
daljinski: -----> Instaliranje python-3.6.2 daljinski:
-----> Instaliranje pip daljinskog upravljača: ----->
Instaliranje zahtjeva s pip-om
...
daljinski: -----> Udaljeno otkrivanje tipova procesa:
                  Profil profila deklarira tipove -> web
daljinski:
daljinski: -----> Kompresija... daljinski:
                  Urađeno: 49,4M
daljinski: -----> Pokretanje...
daljinski: Objavljeno v8 https://
daljinski: <appname>.herokuapp.com/ postavljeno na Heroku
daljinski:
daljinski: Provjera postavljanja... završeno.
Na https://git.heroku.com/<appname>.git * [nova
grana] master -> master
```

Aplikacija je sada raspoređena i radi, ali neće raditi ispravno jer naredba deploy koja inicijalizira tabele baze podataka još nije izvršena. Heroku klijent može pokrenuti ovu naredbu na sljedeći način:

```
$ heroku run flask deploy  
Pokretanje flask deploy na <appname>... up, run.3771 (Besplatno)  
INFO [alembic.runtime.migration] Kontekst impl PostgresqlImpl.  
INFO [alembic.runtime.migration] Pretpostavlja se transakcijski DDL.  
...
```

Nakon što se kreiraju i konfiguiraju tablice baze podataka, aplikacija se može ponovo pokrenuti tako da počne čisto s ažuriranom bazom podataka:

```
$ heroku restart  
Ponovno pokretanje dynos-a na <appname>... gotovo
```

Aplikacija bi sada trebala biti u potpunosti raspoređena i online na <https://<appname>.herokuapp.com>.

Pregledavanje dnevnika

aplikacije Heroku bilježi izlaz evidentiranja koji generira aplikacija. Za pregled sadržaja dnevnika koristite naredbu logs :

```
$ heroku dnevniki
```

Tokom testiranja, takođe može biti zgodno da se „repi“ log fajl, što se može uraditi na sledeći način:

```
$ heroku zapis -t
```

Postavljanje nadogradnje

Kada Heroku aplikaciju treba nadograditi, isti proces treba ponoviti. Nakon što su sve promjene unesene u Git spremište, sljedeće naredbe izvode nadogradnju:

```
$ heroku održavanje:on $ git  
push heroku master $ heroku  
run flask deploy $ heroku  
restart $ heroku održavanje:  
isključeno
```

Opcija održavanja dostupna na Heroku CLI-u će isključiti aplikaciju tokom nadogradnje i prikazat će statičnu stranicu koja obaveštava korisnike da će se stranica uskoro vratiti. Ovo sprječava korisnike da pristupe aplikaciji dok prolazi kroz proces nadogradnje.

Docker kontejneri

Sada ste upoznati sa Herokuom, koji je opcija za postavljanje na prilično visokom nivou. U ovom odeljku ćete naučiti kako da radite sa kontejnerima, a posebno sa Docker platformom, koja nije toliko automatizovana kao PaaS, ali pruža veću fleksibilnost i nije vezana za određenog dobavljača oblaka.

Kontejneri su posebna vrsta virtualnih mašina koje rade na vrhu jezgra operativnog sistema domaćina, za razliku od standardnih virtualnih mašina, koje imaju svoje virtuelizovano jezgro i hardver. Budući da se virtuelizacija zaustavlja na kernelu, kontejneri su mnogo lakši i efikasniji od virtualnih mašina, ali im je potrebna namenska podrška ugrađena u operativni sistem. Linux kernel ima punu podršku za kontejnere.

Instaliranje Docker-

a Najpopularnija platforma kontejnera je **Docker**, koja ima besplatno Community Edition (poznato kao Docker CE) i Enterprise Edition zasnovano na pretplati (Docker EE). Docker se može instalirati na tri glavna desktop operativna sistema, kao i na cloud servere. Najlakši način da razvijete i testirate "kontejneriziranu" aplikaciju je da instalirate Docker CE na vaš razvojni sistem. Za macOS i Microsoft Windows dostupni su programi za instalaciju jednim klikom iz **Docker Store-a**. Ova stranica također uključuje upute za instalaciju za CentOS, Fedora, Debian i Ubuntu Linux distribucije.

Nakon što završite instalaciju Docker CE na vašem sistemu, trebali biste moći pristupiti docker komandi sa vašeg terminala:

```
$ docker verzija
Klijent:
Verzija: API      17.06.0-ce
verzija: 1.30 Go verzija:
go1.8.3 Git commit: 02c1d87
Napravljen: OS/Arch:
                  Pet Jun 23 21:31:53 2017
                  darwin/amd64

Server:
Verzija: API      17.06.0-ce
verzija: 1.30 (minimalna verzija 1.12)
Idi verzija: go1.8.3 Git urezivanje:
02c1d87 Napravljeno: pet, 23.
juna 21:51:55 2017 OS/Arch: amd64 Eksperimentalno: istina
```



Docker za Windows zahtijeva da Microsoftova Hyper-V funkcija bude omogućena. Instalater će ga normalno omogućiti umjesto vas, ali ako se čini da Docker ne radi ispravno nakon instalacije, stanje hipervizora Hyper-V je prva stvar koju treba provjeriti. Trebalo bi da imate na umu da će omogućavanje Hyper-V na vašoj Windows mašini sprečiti rad drugih hipervizora (kao što je Oracleov VirtualBox). Ako vaš sistem ne podržava Hyper-V virtualizaciju ili vam je potrebno Docker rješenje koje ne čini neupotrebljivim druge tehnologije virtualizacije, možda ćete htjeti instalirati **Docker Toolbox**, naslijedjeni Docker proizvod za Windows koji je baziran na VirtualBoxu.

Izrada slike kontejnera Prvi zadatak

kada radite sa kontejnerima je da napravite sliku kontejnera za aplikaciju. Slika je snimak sistema datoteka kontejnera, koji se koristi kao predložak pri pokretanju novih kontejnera. Docker očekuje da upute za kreiranje slike budu dostavljene u datoteci pod nazivom Dockerfile. **Primjer 17-9** prikazuje Dockerfile koji gradi aplikaciju predstavljenu u ovoj knjizi.

Primjer 17-9. Dockerfile: skripta za izgradnju slike kontejnera

```
IZ python:3.6-alpine
```

```
ENV FLASK_APP flasky.py
ENV FLASK_CONFIG docker
```

```
RUN adduser -D flasky
USER flasky
```

```
WORKDIR /home/flasky
```

```
Zahtjevi za KOPIRANJE RUN python
-m venv venv RUN venv/bin/pip install
-r zahtjevi/docker.txt
```

```
COPY app app
COPY migracije migracije COPY
flasky.py config.py boot.sh ./
```

```
# konfiguracija vremena izvođenja
EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

Naredbe izgradnje koje se mogu uključiti u Dockerfile detaljno su dokumentirane u Dockerfile [referenci](#). U suštini, ovo su komande za implementaciju koje instaliraju i konfigurišu aplikaciju u sistemu datoteka kontejnera, koji je izolovan od vašeg sistema.

Naredba FROM je potrebna u svim Dockerles-ovima za specificiranje osnovne slike spremnika od koje se počinje. U većini slučajeva, ovo će biti slika koja je javno dostupna u [Docker Hubu](#), Docker-ovo spremište slika kontejnera. Repozitorijum sadrži zvanične slike za nekoliko verzija Python interpretera. Ovo su slike koje imaju osnovni operativni sistem na kojem je instaliran Python. Slike su navedene sa imenom i oznakom. Ime službene slike Docker Hub Python je jednostavno python. Različite oznake koje su dostupne mogu se vidjeti na stranici Docker Hub za sliku. Za python sliku, oznake se koriste za određivanje željene verzije interpretatora i platforme.

Za ovu aplikaciju, 3.6 interpreter izgrađen na vrhu [Alpine Linuxa](#) koristi se distribucija. Alpine Linux je platforma koja se obično koristi u slikama kontejnera zbog svoje male veličine veličina.



Verzije Docker-a za macOS i Windows mogu pokretati kontejnere zasnovane na Linuxu.

Naredba ENV definira varijable okruženja vremena izvršavanja. Ova komanda uzima dva argumenta: ime varijable i njenu vrijednost. Sve varijable okruženja definirane ovom naredbom bit će dostupne kada se izvrši kontejner baziran na ovoj slici. Ovdje je definirana varijabla FLASK_APP koju zahtijeva naredba flask , kao i FLASK_CONFIG, što je ime klase konfiguracije koju aplikacija koristi da se konfiguriše kada se pokrene. Docker implementacija će koristiti novu konfiguraciju zvanu docker, implementiranu u klasu DockerConfig kao što je prikazano u [primjeru 17-10](#). Ova nova konfiguraciona klasa nasljeđuje ProductionConfig i samo konfigurira evidentiranje da bude usmjeren na stderr, koji Docker automatski hvata i izlaže putem docker logs komande.

Primjer 17-10. cong.py: Docker konguracija

```
class DockerConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)

konfiguracija =
{ # ...
```

```
'docker': DockerConfig, #  
...  
}
```

Naredba RUN izvršava naredbu u kontekstu slike spremnika. Prilikom prvog pojavljivanja RUN-a, unutar kontejnera se kreira flasky korisnik. Naredba adduser dio je Alpine Linuxa i dostupna je u osnovnoj slici odabranoj naredbom FROM . -D argument za adduser potiskuje interaktivni upit za lozinku korisnika.

Naredba USER odabire korisnika pod kojim će se kontejner pokrenuti, kao i korisnika za preostale naredbe u Dockerle-u. Docker podrazumevano koristi root korisnika, ali se smatra dobrom praksom da se prebacite na običnog korisnika kada root pristup nije potreban.

Naredba WORKDIR definira direktorij najviše razine u koji će aplikacija biti instalirana. Za ovu aplikaciju koristi se početni direktorij za novokreiranog korisnika flasky. Preostale komande u Dockerle-u će se izvršiti sa ovim direktorijumom kao trenutnim direktorijumom.

Naredba COPY kopira datoteke iz lokalnog sistema datoteka u sistem datoteka kontejnera. Direktorijumi sa zahtjevima, aplikacijama i migracijama se kopiraju u cijelosti, a zatim se kopiraju i flasky.py, cong.py i nove datoteke boot.sh (o kojima će se uskoro raspravljati).

Dvije dodatne naredbe RUN kreiraju virtualno okruženje i instaliraju zahtjeve u njemu. Za Docker je kreirana posebna datoteka sa zahtjevima kao zahtjevi/docker.txt. Ova datoteka uvozi sve zavisnosti iz requirements/common.txt i dodaje Gunicorn, koji će se koristiti kao web server kao u Heroku implementaciji.

Naredba EXPOSE definira port na koji će aplikacija koja radi unutar kontejnera instalirati svoj server. Kada se kontejner pokrene, Docker će mapirati ovaj port u pravi port na glavnom računaru, tako da kontejner može primati zahtjeve iz vanjskog svijeta.

Konačna komanda je ENTRYPOINT. Ova komanda specificira kako izvršiti aplikaciju kada se kontejner pokrene. Nova datoteka boot.sh, kopirana u gornji kontejner, koristi se kao skripta za pokretanje. **Primjer 17-11** prikazuje sadržaj ove datoteke.

Primjer 17-11. boot.sh: skripta za pokretanje kontejnera

```
#!/bin/sh  
izvorni venv/bin/aktiviraj flask  
deploy exec gunicorn -b  
0.0.0.0:5000 --access-logfile - --error-logfile - flasky:app
```

Skripta počinje aktivacijom venv virtuelnog okruženja koje je kreirano kao dio izgradnje. Zatim pokreće naredbu deploy aplikacije , izgrađenu ranije u ovom poglavlju i također korištenu za Heroku implementaciju. Ovo će kreirati novu bazu podataka, nadograditi je na najnoviju verziju i umetnuti zadane uloge. Pošto varijabla okruženja DATABASE_URL nije postavljena, baza podataka će koristiti SQLite mašinu. Tada se pokreće Gunicorn server koji sluša na portu 5000. Docker hvata sav izlaz iz aplikacije i predstavlja ga kao logove, tako da je Gunicorn konfiguriran da zapisuje i svoje pristupne datoteke i datoteke dnevnika grešaka u standardni izlaz. Pokretanje Gunicorn-a sa exec čini da Gunicorn proces preuzme proces koji pokreće datoteku boot.sh. Ovo je učinjeno zato što Docker posebnu pažnju posvećuje procesu koji pokreće kontejner i očekuje da on bude glavni proces tokom njegovog životnog veka. Kada se ovaj proces završi, završava se i kontejner.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17d da provjerite ovu verziju aplikacije.

Slika kontejnera za Flasky sada se može napraviti na sljedeći način:

```
$ docker build -t flasky: najnovije .
Slanje konteksta izgradnje Docker demonu 51.08MB Korak
1/14 : IZ python:3.6-alpine
--> a6beab4fa70b
...
Uspješno izgrađen 930e17a89b42
Uspješno označen flasky:latest
```

Argument -t za docker build daje ime i oznaku slici kontejnera, odvojene dvotočkom. Najnovije ime oznake se obično koristi za najnoviju verziju slike kontejnera . Tačka na kraju naredbe izgradnje postavlja trenutni direktorij kao direktorij najviše razine tokom izgradnje. Docker će tražiti Dockerfile u ovom direktoriju, a također će učiniti datoteke u ovom direktoriju i sve poddirektorije dostupne za dodavanje u sliku kontejnera.

Kao rezultat uspješne naredbe za izgradnju docker -a, izgrađena slika kontejnera se pohranjuje u lokalno spremište slika. Komanda docker images prikazuje sadržaj spremišta slika na vašem sistemu:

REPOZITORIJ TAG	ID SLIKE KREIRANA	VELIČINA
flasky najnoviji python	930e17a89b42 prije 5 minuta 127MB 88.7MB	
3.6-alpine a6beab4fa70b	sedmice	

Ovaj spisak uključuje upravo napravljenu flasky:najnoviju sliku, kao i osnovnu sliku tumača za Python 3.6 navedenu u FROM Dockerleu, koju Docker preuzima i instalira kao dio izrade.

Pokretanje kontejnera

Kada se napravi slika kontejnera za aplikaciju, ostaje samo da je pokrenete. Naredba docker run čini ovo vrlo jednostavnim zadatkom:

```
$ docker run --name flasky -d -p 8000:5000 \
-e SECRET_KEY=57d40f677aff4d8d96df97223c74d217 \
-e MAIL_USERNAME=<vaše-gmail-korisničko ime> \
-e MAIL_PASSWORD=<vaša-gmail-lozinka> flasky:najnovija
```

Opcija --name daje naziv kontejneru. Imenovanje kontejnera je opciono; ako ime nije dato, Docker ga generiše koristeći nasumično odabrane riječi.

Opcija -d pokreće kontejner u odvojenom režimu, što znači da će kontejner raditi u pozadini na vašem sistemu. Kontejner koji nije odvojen radi se kao zadatak prednjeg plana vezan za sesiju konzole.

Opcija -p mapira port 8000 u host sistemu u port 5000 unutar kontejnera. Docker pruža fleksibilnost mapiranja portova kontejnera na bilo koji port u host sistemu. Ovo mapiranje omogućava da dvije ili više instanci iste slike kontejnera rade na različitim host portovima, dok svaka instanca koristi svoj virtuelizirani port 5000.

Opcija -e definiše varijable okruženja koje će postojati u kontekstu kontejnera, kao dodatak svim varijablama definisanim u vreme izgradnje sa ENV komandom u Dockerle-u. Vrijednost dodijeljena varijabli SECRET_KEY osigurava da su korisničke sesije i tokeni potpisani jedinstvenim ključem koji je vrlo teško pogoditi. Trebali biste generirati vlastiti jedinstveni ključ za ovu varijablu. Vrijednosti za varijable MAIL_USERNAME i MAIL_PASSWORD konfiguiraju slanje e-pošte putem usluge Gmail. Za proizvodnu implementaciju koja koristi drugog dobavljača usluga e-pošte , varijable MAIL_SERVER, MAIL_PORT i MAIL_USE_TLS također treba definirati.

Posljednji argument u docker run komandi je slika kontejnera i oznaka za izvršenje. Ovo bi trebalo odgovarati imenu i oznaci datim kao opcija -t naredbi za izgradnju dockera .

Kada se kontejner pokrene u pozadini, docker run komanda ispisuje ID kontejnera na konzolu. Ovo je 256-bitni jedinstveni identifikator isписан u heksadecimalnom zapisu. Ovaj ID se može koristiti u svim naredbama koje zahtijevaju referencu na kontejner (u praksi je potrebno dati samo prvih nekoliko znakova ID-a, tako da se spremnik može jedinstveno identificirati).

Da biste potvrdili da je kontejner pokrenut, može se koristiti naredba docker ps :

```
$ docker ps
ID SLIKA KONTEJNERA          CREATED      STATUS    LUKE           IMENA
71357ee776ae flasky:zadnje 4 sekunde prije 8 sekundi 0.0.0.0:8000->5000/tcp flasky
```

Pošto je kontejner sada pokrenut i radi, možete pristupiti kontejnerizovanoj aplikaciji na portu 8000 vašeg sistema, bilo lokalno kao <http://localhost:8000> ili sa bilo kog drugog računara u mreži kao <http://<ip-adresa>:8000>.

Da zaustavite ovaj kontejner, koristite naredbu docker stop :

```
$ docker stop 71357ee776ae
71357ee776ae
```

Naredba stop zaustavlja kontejner, ali ga ne uklanja iz sistema. Da biste ga uklonili, koristite naredbu docker rm :

```
$ docker rm 71357ee776ae
71357ee776ae
```

Ove dvije operacije se mogu kombinirati u jednu pomoću docker rm -f:

```
$ docker rm -f 71357ee776ae
71357ee776ae
```

Provjera pokrenutog kontejnera Kada

izgleda da se kontejner ne ponaša, možda će biti potrebno da ga otklonite greške. Najočigledniji mehanizam za otklanjanje grešaka je dodavanje izjava za evidentiranje u aplikaciju, a zatim praćenje pokrenutog kontejnera pomoću naredbe docker logs .

U nekim situacijama, međutim, može biti zgodnije otvoriti sesiju ljske na pokrenutom kontejneru kako bi se mogla pažljivije pregledati. Naredba docker exec ovo omogućava:

```
$ docker exec -it 71357ee776ae sh
```

U ovom primeru, Docker će otvoriti sesiju ljske sa sh (Unix školjka) bez prekidanja kontejnera. Opcije -it povezuju terminalsku sesiju iz koje se izdaje naredba sa novim procesom, tako da se ljskom može interaktivno raditi. Ako kontejner uključuje druge, naprednije školjke kao što je bash ili čak Python interpreter, oni se također mogu koristiti.

Uobičajena strategija pri rješavanju problema sa kontejnerima je kreiranje posebne slike učitane dodatnim alatima kao što je debugger koji se kasnije može pozvati iz ljske sjednici.

Prebacivanje slike vašeg kontejnera u eksterni registar Imati sliku kontejnera lokalno je zgodno kada se razvija i testira aplikacija, ali kada ste spremni da podelite sliku sa drugima, morate da je gurnete na spoljni server registra.

Docker Hub registar je Docker-ovo spremište slika, zgodna usluga na kojoj možete hostirati svoje slike. Besplatni Docker Hub nalog omogućava vam da pohranite neograničen broj slika javnih kontejnera, ali samo jednu privatnu sliku. Plaćeni planovi povećavaju broj privatnih slika koje možete ugostiti. Da kreirate svoj Docker Hub nalog, idite na <https://hub.docker.com>.

Nakon što imate Docker Hub nalog, možete se prijaviti na njega iz komandne linije pomoću docker login naredbe:

```
$ docker login
```

Prijavite se sa svojim Docker ID-om da biste gurnuli i povukli slike iz Docker Hub-a.

Korisničko ime: <your-dockerhub-username>

Lozinka: <your-dockerhub-password> Prijava
je uspjela



Da biste se prijavili u spremište slika kontejnera koje nije Docker Hub, pronesite adresu svog spremišta kao argument za prijavu na docker.

Lokalne slike kontejnera dobijaju jednostavno ime. Da biste se pripremili za prosljeđivanje slike u Docker Hub, naziv slike mora imati prefiks naziva Docker Hub računa i kosu crtu kao razdjelnik. Flasky :najnovija slika napravljena ranije može dobiti sekundarno ime pravilno formatirano za prosljeđivanje u Docker Hub pomoću naredbe docker tag :

```
$ docker oznaka flasky: najnovije <your-dockerhub-username>/flasky: najnovije
```

Za otpremanje slike u Docker Hub, koristite docker push naredbu:

```
$ docker push <your-dockerhub-username>/flasky:latest
```

Slika kontejnera je sada javno dostupna i svako može pokrenuti kontejner na osnovu nje pomoću docker run komande:

```
$ docker run --name flasky -d -p 8000:5000 \ <vaše-  
dockerhub-korisničko ime>/flasky: najnovije
```

Korištenje vanjske baze podataka

Jedan nedostatak načina na koji je Flasky bio raspoređen kao Docker kontejner je to što zadana SQLite baza podataka živi u istom kontejneru kao i aplikacija. Ovo otežava izvođenje nadogradnje, jer jednom kada se pokrenuti kontejner zaustavi, baza podataka nestaje s njim.

Bolji pristup je da se server baze podataka hostuje odvojeno od kontejnera aplikacije. To čini nadogradnju aplikacije uz očuvanje baze podataka lakis zadatkom, jer sve što je potrebno je zamijeniti kontejner aplikacije novim.

Docker promoviše modularni pristup izgradnji aplikacije, u kojoj se svaki servis nalazi u svom vlastitom kontejneru. Dostupne su slike javnih kontejnera za MySQL, Postgres i mnoge druge servere baza podataka. Docker run komanda se može koristiti za implementaciju bilo koje od njih direktno na vaš sistem. Sljedeća naredba postavlja MySQL 5.7 server baze podataka na vaš sistem:

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=da \
-e MYSQL_DATABASE=flasky -e MYSQL_USER=flasky -e MYSQL_PASSWORD=<lozinka-
baze podataka> mysql/mysql7.
```

Ova komanda kreira kontejner pod imenom mysql koji radi u pozadini. Opcija -e dodjeljuje nekoliko varijabli okruženja koje ovaj kontejner uzima kao konfiguraciju.

Ove i mnoge druge varijable su dokumentovane na stranici Docker Hub za MySQL sliku. Prethodna naredba konfigurira bazu podataka nasumično generiranom root lozinkom (koristite docker logove mysql odmah nakon pokretanja kontejnera da vidite dodijeljenu lozinku u evidenciji) i potpuno novom bazom podataka koja se zove flasky koja je konfiguirirana da joj pristupi korisnik takođe nazvan flasky. Morate dati sigurnu lozinku za ovog korisnika kao vrijednost za varijablu okruženja MYSQL_PASSWORD .

Da bi se mogao povezati na MySQL bazu podataka, SQLAlchemy zahtijeva da se instalira podržani MySQL klijentski paket kao što je pymysql . Ovaj paket se može dodati u datoteku zahtjeva docker.txt.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17e da provjerite ovu verziju aplikacije.

Promjena napravljena u datoteci requirements/docker.txt zahtijeva da se slika kontejnera ponovo izgradi:

```
$ docker build -t flasky: najnovije .
```

Ako još uvijek koristite prethodni kontejner aplikacije, zaustavite ga i uklonite pomoću docker rm -f. Zatim pokrenite novi kontejner s ažuriranom aplikacijom:

```
$ docker run -d -p 8000:5000 --link mysql:dbserver \
-e DATABASE_URL=mysql+pymysql://flasky:<lozinka-baze podataka>@dbserver/flasky \
-e MAIL_USERNAME=<vaše-gmail-korisničko ime> -e MAIL_PASSWORD=<vaše-gmail-lozinka> \
flasky:najnovija
```

Ovdje su prikazana dva dodatka docker run komandi. --link opcija konfiguriše vezu između novog kontejnera i drugog postojećeg. Argument za --link sastoji se od dva imena odvojena dvotočkom: naziv izvornog spremnika ili ID i alias za taj spremnik u spremniku koji se kreira. U ovom primjeru izvorni kontejner je mysql, kontejner baze podataka je pokrenut ranije. Ovaj kontejner će biti dostupan u novom Flasky kontejneru sa dbserver imenom hosta.

Da bi se dovršila konfiguracija, dodaje se varijabla okruženja DATABASE_URL , sa URL-om veze koja ukazuje na flasky bazu podataka u mysql kontejneru. Dbserver alias se koristi kao host baze podataka, jer Docker osigurava da se ovo ime razluči na IP adresu povezanog kontejnera . Vrijednost varijable okruženja MYSQL_PASSWORD postavljena u mysql kontejner mora biti uključena i u URL veze za ovaj kontejner. Vrijednost DATABASE_URL nadjačava zadanu SQLite bazu podataka, tako da će ovom jednostavnom promjenom kontejner biti konfiguriran da se poveže s MySQL bazom podataka.



Docker Hub spremište je zlatni rudnik veoma korisnih aplikacija i usluga koje su upakovane i spremne za upotrebu u Docker okruženju, bilo samostalno ili kao osnovne slike za vaše sopstvene kontejnere. Otkrićete da sve vrste projekata (uključujući baze podataka, web servere, balansere opterećenja, programske jezike, operativne sisteme i više) nude zvanične slike.

Orkestracija kontejnera sa Docker Compose Kontejnerske

aplikacije se obično sastoje od nekoliko pokrenutih kontejnera. U prethodnom odeljku ste videli da glavna aplikacija i server baze podataka rade u nezavisnim kontejnerima. Kako aplikacija postaje sve složenija, uvjek će joj trebati više kontejnera. Neke aplikacije će zahtijevati dodatne usluge, kao što su redovi poruka ili keš memorije. Druge aplikacije mogu iskoristiti prednosti arhitekture mikro-servisa i imati distribuiranu strukturu sa nekoliko manjih sub aplikacija, od kojih svaka radi u svom kontejneru. Aplikacije koje moraju podnijeti velika opterećenja ili moraju biti tolerantne na greške htjet će se smanjiti pokretanjem nekoliko instanci iza balansera opterećenja.

Kako se broj kontejnera koji su dio aplikacije povećava, zadatak upravljanja i koordinacije svih ovih kontejnera će postati mnogo teži ako se koristi samo Docker. Okviri orkestracije kontejnera izgrađeni na vrhu Dockera pomažu u ovom zadatku.

Skup alata Compose pruža osnovnu funkcionalnost orkestracije, uključenu u instalaciju Dockera. Sa Compose, kontejneri koji su dio aplikacije opisani su u konfiguracijskoj datoteci, obično nazvanoj docker-compose.yml. Naredba docker compose tada može pokrenuti sve kontejnere povezane s aplikacijom pomoću jedne naredbe.

Primer 17-12 prikazuje datoteku docker-compose.yml koja predstavlja kontejnerski Flasky zajedno sa njegovim MySQL servisom.

Primjer 17-12. docker-compose.yml: sastaviti konguraciju

verzija: '3' usluge:

```
flasky:
  build: .
  portovi:
    - "8000:5000"
  env_file: .env
  linkovi:
    - mysql:dbserver
  restart: uvijek mysql:
image: "mysql/mysql-
server:5.7" env_file: .env-mysql restart: uvijek
```

Ova datoteka je napisana u YAML-u, koji je čist i jednostavan format koji može predstavljati hijerarhijske strukture koje se sastoje od mapa ključ/vrijednost i lista. Ključ verzije određuje koja se verzija Composea koristi, a servisni ključ definira kontejnere aplikacije kao njene potomke. U slučaju Flaskyja, to su dva servisa pod nazivom flasky i mysql.

Za usluge kao što je flasky, koje su izgrađene kao dio aplikacije, potključevi specificiraju argumente koji se daju naredbama za izgradnju i docker run . Ključ izgradnje specificira direktorij izgradnje, gdje se nalazi Dockerle .

Ključ portova specificira mapiranja mrežnih portova . Env_file ključ je zgodan način za definiranje nekoliko varijabli okruženja koje su potrebne kontejneru. Ključ links uspostavlja vezu do MySQL kontejnera, izlažući ga sa imenom hosta dbserver. Ključ za ponovno pokretanje postavljen na uvijek pruža jednostavan način za Docker da automatski ponovo pokrene kontejner ako neočekivano izađe. Datoteka .env za ovu primenu treba da sadrži sledeće varijable:

```
FLASK_APP=flasky.py
FLASK_CONFIG=docker
SECRET_KEY=3128b4588e7f4305b5501025c13ceca5
MAIL_USERNAME=<vaše-gmail-korisničko ime>
MAIL_PASSWORD=<vaš-gmail-password>data-gmail-password
@lamsqai-password>datamsqaiaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
```

Mysql servis ima jednostavniju strukturu, jer se radi o servisu koji se pokreće sa slike zaliha za koju nije potreban korak izgradnje. Ključ slike specificira ime i oznaku slike kontejnera za korištenje za ovu uslugu. Kao i kod naredbe Docker run , Docker će preuzeti ovu sliku iz registra slika kontejnera. Env_file i ključevi restart su slični onima koji se koriste u flasky kontejneru. Obratite pažnju na to kako su varijable okruženja za MySQL kontejner pohranjene u zasebnoj datoteci pod nazivom .env-mysql. Iako bi bilo lakše dodati varijable okruženja koje su potrebne svim kontejnerima u .env datoteku, dobra je praksa spriječiti da jedan kontejner ima pristup tajnama drugog. Datoteci .env-mysql treba definirati sljedeće varijable okruženja:

```
MYSQL_RANDOM_ROOT_PASSWORD=da
MYSQL_DATABASE=flasky
MYSQL_USER=flasky
MYSQL_PASSWORD=<lozinka-baze podataka>
```



Datoteke .env i .env-mysql sadrže lozinke i druge osjetljive informacije, tako da ih nikada ne treba dodavati u kontrolu izvora.



Kompletna referenca za datoteku docker-compose.yml nalazi se na "[Docker web stranici](#)".

Tipičan problem sa orkestriranim sistemima je taj što se kontejneri pokreću pogrešnim redosledom—ili ispravnim redosledom, ali bez davanja dovoljno vremena kontejnerima za osnovne usluge da se pokrenu i inicijalizuju pre pokretanja kontejnera višeg nivoa koji zavise od njih. U slučaju Flaskyja, mysql kontejner mora da se pokrene prvi, tako da baza podataka bude u funkciji kada se flasky kontejner pokrene. Zatim se može povezati s bazom podataka, primjeniti migracije baze podataka i konačno pokrenuti web server.

Compose će pokrenuti mysql i flasky kontejnere u pravom redosledu, jer će otkriti zavisnost između njih iz ključa linkova u flasky kontejneru. Ali Compose neće čekati da se MySQL pokrene, što bi moglo potrajati nekoliko sekundi.

Prilikom dizajniranja distribuiranih sistema, dobra je praksa implementirati ponovne pokušaje u svim konekcijama na eksterne usluge. [Primjer 17-13](#) pokazuje kako boot.sh skripta to

startovanja flasky kontejner se može učiniti robusnjim ponovnim pokušajem naredbe flask deploy , koja ponavlja nadogradnju baze podataka dok ne uspije.

Primjer 17-13. boot.sh: čeka da se baza podataka pokrene

```
#!/bin/sh
izvor venv/bin/aktiviraj

dok je istinito; do
    flask
    implementirati ako [[ "$?" ==
        "0" ]]; onda prekid
    biti
    Naredba echo Deploy nije uspjela, ponovni pokušaj za 5 sekundi...
    spavanje 5 završeno

exec gunicorn -b :5000 --access-logfile - --error-logfile - flasky:app
```

Pokretanjem flask deploy unutar petlje ponovnog pokušaja, kontejner će moći tolerirati kvarove zbog toga što usluga baze podataka nije odmah spremna da prihvati zahtjeve.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17f da provjerite ovu verziju aplikacije. Također provjerite da li su .env i .env-mysql datoteke okruženja kreirane i popunjene vrijednostima ispravnim za vaše okruženje. ment.

Sada kada je konfiguracija Compose završena, aplikacija se može pokrenuti naredbom docker-compose up :

```
$ docker-compose up -d --build
```

--build opcija za docker-compose up označava da se korak izgradnje treba pokrenuti prije pokretanja aplikacije. Ovo će uzrokovati stvaranje slike flasky kontejnera. Nakon što je slika kreirana, mysql i flasky kontejneri će se pokrenuti tim redoslijedom. Opcija -d pokreće kontejnere u odvojenom režimu, kao i sa pojedinačnim kontejnerom. Nakon nekoliko sekundi, aplikacija bi trebala biti pokrenuta i pokrenuta u pozadini i trebali biste se moći povezati s njom na <http://localhost:8000>.

Compose konsoliduje evidenciju iz svih kontejnera u jedan tok, što možete vidjeti pomoću naredbe docker-compose logs :

```
$ docker-compose logs
```

Ili, ako želite stalno pratiti tok dnevnika:

```
$ docker-compose logs -f
```

Naredba docker-compose ps prikazuje sažetak svih kontejnera aplikacije koji su pokrenuti i njihovo stanje:

Ime	Zapovjedi	Država	Luke
flasky_flasky_1 ./boot.sh flasky_mysql_1 Gore /entrypoint.sh mysqld Gore		0.0.0.0:8000->5000/tcp 3306/ tcp, 33060/tcp	

Da biste nadogradili aplikaciju na novu verziju, jednostavno napravite potrebne promjene na njoj i ponovite naredbu docker-compose up koja je prethodno korištena za pokretanje. Compose će ponovo izgraditi kontejner aplikacije ako se nešto promijeni, a zatim zamijeniti stariji kontejner novim.

Da zaustavite aplikaciju, koristite naredbu docker-compose down ili docker-compose rm --stop --force ako također želite ukloniti zaustavljene kontejnere.

Čišćenje starih kontejnera i slika Dok radite sa

kontejnerima, vaš sistem će uvek akumulirati stare kontejnere ili slike koje više nisu potrebne. Dobra je ideja da ih redovno pregledavate i čistite kako ne bi zauzimali prostor na sistemu.

Da vidite listu kontejnera u sistemu, koristite sljedeću naredbu:

```
$ docker ps -a
```

Ovo će prikazati kontejnere koji su pokrenuti i kontejnere koji su zaustavljeni, ali su još uvijek u sistemu. Da biste izbrisali bilo koji kontejner sa ove liste, koristite naredbu docker rm -f i navedite imena ili ID-ove za uklanjanje:

```
$ docker rm -f <ime-ili-id> <ime-ili-id> ...
```

Da vidite listu slika kontejnera pohranjenih u vašem sistemu, koristite naredbu docker images . Ako postoje slike koje želite ukloniti, to možete učiniti pomoću naredbe docker rmi .

Neki kontejneri kreiraju virtuelne volumene na glavnom računaru koji se koriste za skladištenje izvan sistema datoteka kontejnera. Slika MySQL kontejnera, na primjer, stavlja sve datoteke baze podataka u volumen. Možete vidjeti listu svih dodijeljenih volumena u vašem sistemu pomoću docker volume ls. Da biste uklonili volumen koji se ne koristi, koristite docker volume rm.

Ako više volite automatsko čišćenje, docker sistemska komanda prune --volumes će ukloniti sve neiskorištene slike ili volumene i sve zaustavljene kontejnere koji su još uvijek u sistemu.

Upotreba Dockera u proizvodnji

Mnogi ljudi smatraju Docker samo platformom za razvoj i testiranje. Iako se tehnike predstavljene u prethodnim odjeljcima mogu koristiti za implementaciju aplikacija na proizvodnim serverima koji rade na Dockeru, postoje neka ograničenja i sigurnosni problemi koje treba uzeti u obzir:

Nadgledanje i upozorenje

Šta se dešava ako se sruši kontejnerska aplikacija? Docker može ponovo pokrenuti kontejner koji neočekivano izađe, ali neće nadzirati vaše kontejnere, niti će slati upozorenja kada se ponašaju nepravilno.

Docker

za evidentiranje održava poseban tok dnevnika za svaki kontejner. Compose ovo poboljšava nudeći konsolidovani tok, ali bez mogućnosti dugotrajnog skladištenja ili pretraživanja i filtriranja.

Upravljanje tajnama

Konfiguriranje lozinki i drugih vjerodajnica putem varijabli okruženja je nesigurno, budući da Docker izlaže unaprijed konfiguirane varijable okruženja putem naredbe docker inspect ili preko svog API-ja.

Pouzdanost i skaliranje

Da bi se pomoglo u toleranciji grešaka, ili da bi se prilagodili rastućim zahtjevima opterećenja, potrebno je pokrenuti nekoliko instanci aplikacije na nekoliko hostova i iz jednog ili više balansera opterećenja.

Ova ograničenja se generalno rješavaju razrađenijim okvirima orkestracije koji su izgrađeni na vrhu Docker-a ili drugih vremena izvođenja kontejnera. Okviri kao što su Docker Swarm (sada ugrađen u Docker), Apache Mesos i Kubernetes su dobar izbor za izgradnju robusnih implementacija kontejnera.

Tradicionalne implementacije

Do sada ste vidjeli kako Heroku i Docker upravljaju implementacijama. Da bismo završili ovaj pregled strategija implementacije, ovaj odjeljak će opisati tradicionalnu opciju hostinga, koja uključuje kupovinu ili iznajmljivanje servera, bilo fizičkog ili virtuelnog, i ručno postavljanje svih potrebnih komponenti na njemu. Ovo je očigledno najzahtevnija opcija od svih, ali može biti zgodna opcija kada imate terminalski pristup hardveru proizvodnog servera. Sljedeći odjeljci će vam dati ideju o poslu koji je uključen.

Podešavanje servera

Postoji nekoliko administrativnih zadataka koji se moraju izvršiti na serveru prije nego što može ugostiti aplikacije:

- Instalirajte server baze podataka kao što je MySQL ili Postgres. Korištenje SQLite baze podataka je također moguće, ali se ne preporučuje za proizvodni poslužitelj zbog brojnih ograničenja u pogledu modifikacije postojećih shema baze podataka.
- Instalirajte Mail Transport Agent (MTA) kao što je Sendmail ili Postx za slanje e-pošte korisnicima. Upotreba Gmail-a u produkcijskoj aplikaciji nije moguća, jer ova usluga ima vrlo restriktivne kvote i posebno zabranjuje komercijalnu upotrebu u svojim uvjetima korištenja.
- Instalirajte web server spremjan za proizvodnju kao što je Gunicorn ili uWSGI. • Instalirajte uslužni program za praćenje procesa kao što je Supervizor, koji odmah ponovo pokreće web server ako se sruši ili nakon što se host isključi. • Instalirajte i konfigurišite SSL sertifikat da omogućite siguran HTTP. • (Opcionalno, ali se preporučuje) Instalirajte front-end reverse proxy web server kao što je nginx ili Apache. Ovaj server je konfiguriran da direktno opslužuje statičke datoteke i proslijeđuje zahtjeve aplikacije na web server aplikacije, koji sluša na privatnom portu na lokalnom hostu.
- Osigurajte server. Ovo uključuje nekoliko zadataka koji imaju za cilj smanjenje ranjivosti na serveru, kao što su instaliranje zaštitnih zidova, uklanjanje neiskorištenog softvera i usluga i tako dalje.



Umjesto ručnog izvršavanja ovih zadataka, kreirajte skriptiranu implementaciju koristeći okvir za automatizaciju kao što je Ansible, Chef ili Puppet.

Uvoz varijabli okruženja Slično kao kod Heroku-a i

Docker-a, aplikacija koja radi na samostalnom serveru oslanja se na određene postavke kao što su URL veze sa bazom podataka, vjerodajnice servera e-pošte, itd. koje se nalaze u varijablama okruženja.

Budući da ne postoji Heroku ili Docker za konfigurisanje ovih varijabli prije pokretanja aplikacije, procedura za postavljanje varijabli ovisi o platformi i korištenim alatima. Da bi konfiguracija varijabli okruženja bila lakša i ujednačena na platformama za implementaciju, blok kratkog koda u [primjeru 17-14](#) uvozi u okruženje .env datoteku sličnu onoj koja se koristi s heroku local i dockerom.

sastavite komande, koristeći Python paket koji se zove python-dotenv koji treba da se instalira sa pip. Ovo se radi u flasky.py prije kreiranja instance aplikacije, tako da do trenutka uvoza konfiguracije ove varijable budu dostupne u okruženju.

Primjer 17-14. flasky.py: uvoz okruženja iz .env le

```
import os
from dotenv import load_dotenv

dotenv_path = os.path.join(os.path.dirname(__file__), '.env') ako
os.path.exists(dotenv_path): load_dotenv(dotenv_path)
```

Datoteka .env može definirati varijablu FLASK_CONFIG koja bira konfiguraciju za korištenje, vezu DATABASE_URL , vjerodajnice servera e-pošte, itd. Kao što je ranije objašnjeno, .env fajl ne bi trebao biti dodavan u kontrolu izvora zbog osjetljive prirode nekih od stavke u njemu.



Ako ste kreirali .env datoteku za upotrebu sa Herokuom ili Dockerom, pregledajte je i prilagodite je na odgovarajući način, jer će sa upravo napravljenim promjenama aplikacija uvesti varijable definirane u ovoj datoteci za sve konfiguracije.

Podešavanje evidencije

Za servere bazirane na Unixu, evidentiranje se može poslati syslog daemonu. Nova konfiguracija specifično za Unix može se kreirati kao podklasa ProductionConfig, kao što je prikazano u [Primeru 17-15.](#)

Primjer 17-15. cong.py: Unix primjer konfiguracije

```
klasa UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to syslog
        import logging
        from logging.handlers import SysLogHandler
        syslog_handler = SysLogHandler()
        syslog_handler.setLevel(logging.WARNING)
        app.logger.addHandler(syslog_handler)
```

Sa ovom konfiguracijom, zapisnici aplikacije će biti upisani u konfigurisanu datoteku syslog poruka, obično /var/log/messages ili /var/log/syslog u zavisnosti od Linux distri-

bution. Servis syslog može se konfigurirati da piše zasebnu datoteku dnevnika za dnevnike aplikacije ili da po želji pošalje dnevnike na drugu mašinu.



Ako ste klonirali Git spremište aplikacije na GitHub-u, možete pokrenuti git checkout 17g da provjerite ovu verziju aplikacije.

Dodatni resursi

Gotovo ste završili s ovom knjigom. Čestitamo! Nadam se da su vam teme koje sam pokrio dale solidnu osnovu da počnete da pravite sopstvene aplikacije sa Flaskom. Primjeri koda su otvorenog koda i imaju dopuštenu licencu, tako da ste dobrodošli da koristite onoliko mog koda koliko želite za početak vaših projekata, čak i ako su komercijalne prirode. U ovom kratkom završnom poglavlju želim vam dati listu dodatnih savjeta i resursa koji bi mogli biti korisni dok nastavite raditi sa Flaskom.

Korištenje integriranog razvojnog okruženja (IDE)

Razvijanje Flask aplikacija u integrisanom razvojnem okruženju (IDE) može biti veoma zgodno, jer funkcije kao što su dovršavanje koda i interaktivni debager mogu značajno ubrzati proces kodiranja. Neki od IDE-ova koji dobro rade sa Flaskom su navedeni ovdje:

PyCharm

IDE iz JetBrainsa sa Community (besplatno) i Professional (plaćenim) izdanjima, oba kompatibilna sa Flask aplikacijama. Dostupno na Linux, macOS i Windows.

Visual Studio Code

IDE otvorenog koda iz Microsofta. Python dodatak treće strane mora biti instaliran da biste imali pristup funkcijama dovršavanja koda i otklanjanja grešaka sa Flask aplikacijama. Dostupno na Linuxu, macOS-u i Windowsu.

PyDev

IDE otvorenog koda baziran na Eclipseu. Dostupno na Linuxu, macOS-u i Windowsu.

Pronalaženje flask ekstenzija

Primjeri u ovoj knjizi oslanjaju se na nekoliko ekstenzija i paketa, ali postoji mnogo više onih koji su također korisni i o kojima se nije raspravljalo. Slijedi kratka lista nekih dodatnih paketa koje vrijedi istražiti:

- **Flask-Babel:** Podrška za internacionalizaciju i lokalizaciju • **Marshmallow:**

Serijalizacija i deserializacija Python objekata, korisni za predstavljanje API resursa • **Celery:** Red zadataka za obradu pozadinskih poslova • **Frozen-Flask:** Konverzija Flask aplikacije u statičnu web stranicu • **Flask-DebugToolbar:** Alati za oticanje grešaka u pregledaču • **Flask-aktivne:** Spajanje, minimiziranje i kompajliranje CSS i JavaScript sredstava • **Flask-Session:** Alternativna implementacija korisničkih sesija koje koriste serversku stranu

skladištenje

- **Flask-SocketIO:** Implementacija Socket.IO servera sa podrškom za WebSocket i dugotrajno glasanje

Ako funkcionalnost koja vam je potrebna za vaš projekat nije pokrivena nijednom od ekstenzija i paketa spomenutih u ovoj knjizi, onda bi vaše prvo odredište za traženje dodatnih ekstenzija trebalo biti službeni [register proširenja Flask](#). Druga dobra mjesta za pretraživanje su [Python Package Index](#), [GitHub](#), i [Bitbucket](#).

Dobivanje pomoći

Ako dođete do tačke u kojoj ste blokirani problemom koji ne možete sami riješiti, imajte na umu da postoji zajednica Flask programera poput vas koja će vam rado pomoći.

Sjajno mjesto za postavljanje pitanja o Flasku ili bilo kojim srodnim proširenjima je [Stack Overflow](#).

Drugi programeri koji vide vaše pitanje i znaju kako odgovoriti objavit će svoje odgovore, za koje se glasa za gore ili protiv u zavisnosti od njihovog kvaliteta. Vi, kao vlasnik pitanja, tada možete odabrati najbolji odgovor. Sva pitanja i njihovi odgovori ostaju na stranici i pojavljuju se u rezultatima pretraživanja. Dakle, postavljanjem pitanja na ovoj platformi pomažete u povećanju zbirke informacija o Flasku.

Reddit takođe ima prijateljski [subreddit posvećen Flasku](#) gdje možete postavljati pitanja.

Konačno, ako koristite IRC, #pocoo kanal na Freenodeu posjećuju Flask programeri svih nivoa koji vam mogu pomoći jedan na jedan s vašim problemom.

Uključivanje u Flask

Flask ne bi bio tako sjajan kao što jeste bez posla koji je obavila njegova zajednica programera. Kako sada postajete dio ove zajednice i imate koristi od rada tolikog broja volontera, trebali biste razmisliti o pronalaženju načina da nešto vratite. Evo nekoliko ideja koje će vam pomoći da započnete:

- Pregledajte dokumentaciju za Flask ili vaš omiljeni srodnji projekat i pošaljite ispravke ili poboljšanja.
- Prevedite dokumentaciju na novi jezik. • Odgovarajte na pitanja na stranicama za pitanja i odgovore kao što je [Stack Overflow](#). • Razgovarajte o svom radu sa svojim kolegama na sastancima grupe korisnika ili konferencijama. • Doprinesite ispravke grešaka ili poboljšanja paketa koje koristite. • Napišite nove ekstenzije Flask i pustite ih kao open source. • Objavite svoje aplikacije kao open source.

Nadam se da ćete se odlučiti za volontiranje na jedan od ovih načina, ili bilo koji drugi koji vam je od značaja. Ako jeste, hvala!

Indeks

- Funkcija prekida, 22, 231
- apsolutni URL (u linkovima), 37
- potvrda naloga, 118-125 generisanje
 - tokena za potvrdu sa svojim- opasnim, 118
 - slanje e-poruka potvrde, 120-124 upravljanje nalogom, 125 dodela uloga korisnicima, 135
 - aktivacija, virtuelna okruženja, 4 uloge administratora, 127 dodjela, 131 administrator, uređivač korisničkog profila za, 143-145 admin_required dekorater, 145 after_app_request hook, 238
 - after_request hook, 20
- Alembičke migracije baze podataka, 74
- Alpine Linux, 258
- Prilagođena klasa AnonymousUser, 132
- Mjeseci Apača, 270
- API-ji (aplikacioni programski interfejsi), 199-218
 - uvod u REST, 199-203
 - resursi, 200 web servisa za verzije, 202 RESTful web servisa sa Flask, 203-218 app.add_url_rule metoda, 8, 19 app.cli.command dekorator, 96 app.config objekat, 44 app.run metoda, 11 app.shell_context_processor dekorator 72 kontekst aplikacije, 18 i e-pošta na pozadinskoj niti, 83 direktorij aplikacije, kreiranje, 2
- instanca aplikacije, 7
- struktura aplikacije, osnovna, 7-23
 - opcije komandne linije, 15
 - kompletnih primjera aplikacije, 9 način za oticanje grešaka, 13 razvojni server, 10 dinamičkih ruta, 12
- Flask ekstenzije, 23
- inicijalizacijske aplikacije, 7
- ciklusa zahtjev-odgovor, 17-22 rute i funkcije pregleda, 8
- aplikacije, velika, struktura, 85-97 paket aplikacija, 88-92 implementacija
 - funkcionalnosti aplikacije u nacrtu, 90-92 korištenjem tvornice aplikacije, 88-89 skripta aplikacije, 93 opcije konfiguracije, 86-88 postavljanje baze podataka, 96 struktura projekta, 85 datoteka sa zahtjevima, 93 pokretanje aplikacije, 97 jediničnih testova, 94 dekorater app_errorhandler, 91, 205 tablica asocijacije (baza podataka), 65, 172 tabela pratićaca kao model, 174 auth.login_required dekorater, 208 potvrda autentifikacije1-125 naloga, 118-125 upravljanje nalogom, 125 kreiranje nacrt za, 105 Flask ekstenzije za, 101 Flask-Mail Gmail nalog, 80 u API nacrtu, 204

- registracija novih korisnika, 115-118
sigurnost lozinkom, 102-105 bazirana
na tokenu, 208-210 autentifikacija
korisnika sa Flask-Login, 107-115
- sa Flask-HTTPEAuth, 206 sa Heroku
nalogom, 245 okvira za
automatizaciju, 271 avatare, 146-149
avatare na stranici profila, 147 autora
blog postova, 153
- Argumenti niza upita Gravatar, 146
Gravatar generiranje URL-a, 146
AWS EC2, 243
- B**
- pozadinski poslovi, za slanje e-pošte, 84
pozadinska nit, e-pošta uključena, 83 bash, 2
before_app_request dekorater, 122, 138
before_first_request kuka, 20 before_request
kuka, 20, 122 before_request rukovalac sa
autentifikacijom,
- 208**
- Bleach paket, 164 bloka
(u osnovnim predlošcima), 29
dostupnih blokova u Flash-Bootstrap-u, 32 blog
posta, 151-169 urednik za, 167-169 na stranicama
profila, 154 duge liste sa paginiranjem, 155-161
stalni linkovi na, 165 -167 upita praćenih postova
pomoću pridruživanja bazi podataka, 181-183
postova obogaćenog teksta sa Markdownom i
Flask
- 247**
- konfigurisanje e-pošte za aplikacije na Heroku, 248
- Docker implementacija, 258 za
sporo izvještavanje o upitima, 238
velikih aplikacija, opcije za, 86-88 slanje e-
pošte za greške u aplikaciji, 242
Serveri bazirani na Unixu, evidentiranje,
272 kontejnera, 244, 256
(vidi i Docker)
- pregovaranje sadržaja, 205
kontekstualnih procesora, 134
konteksta, 17 za e-poštu na
pozadinskoj niti, 83 kontekst ljske,
dodavanje, 72 kontrolne strukture (Jinja2),
28 kolačića na strani klijenta, korisničke sesije u,
52 postavke u objektu odgovora , 21 prikazuje
propraćene objave, 184
- Koordinirano univerzalno vrijeme (UTC), 38 alat
za pokrivenost, 222 (vidi također izvještaje o
pokrivenosti koda)

create_app fabrička funkcija koja prilaže nacrt autentifikacije aplikaciji, **106**

Napadi krivotvorena zahtjeva na više lokacija (CSRF), **44**

CSS

- Bootstrap CSS datoteke, **30, 33**
- Bootstrap klase paginacije, **159** klasa za avatar na stranici profila, **147** stilova za blog postove, **153** cURL, **217** varijabla tekuće_vrijeme, **39** trenutna_user kontekstualna varijabla, **113, 142**
- current_user.can funkcija, **132** funkcija current_user.can, **132** funkcija , **132**
- current_user.is_authenticated property, **110,** **114**
- current_user._get_current_object metoda, **152** Cygwin, **2**

D

baza podataka (u URL-ovima baze podataka), **61** baza podataka, **57-77**

- kreiranje tabela, **66** brisanje redova, **68**
- Flask podrška za, **xi**
- umetanje redova, **66**
- integracija sa Python školjkom, **72** velike aplikacije koje koriste različite baze podataka, **87**
- evidentiranje sporih performansi, **237-239**
- pravljenje korisnika svojim sljedbenicima u bazi podataka, **186** upravljanje sa Flask-SQLAlchemy, **61** migracija sa Flask-Migrate, **73-77** dodavanje više migracija, **76** nadogradnja baze podataka, **75** definicija modela, **62-64** modifikacija redovi, **68**

NoSQL, **58**

korisničkih uloga, **127-131**

- dodavanje novih uloga u sesiji ljske, **134**

Model posta za blog postove, **151**

- Rukovanje tekstom Markdown, **164**
- obezbjedivanje baze podataka na Herokuu, **246**
- Python okviri baze podataka, **59-60** upiti praćenih postova pomoću pridruživanja bazi podataka, **181-183**
- filter_by filter upita, **182** pridruži filter upita, **182**

upiti redovi, **68** relacijski model, **57** relacija u, **64-65,** **171-178** tablica asocijacija, **172**

- reprezentacija komentara na blogu, **189-191**

postavljanje u velikoj aplikaciji, **96** SQL, **57** SQL naspram NoSQL, **59** pohranjivanje MD5 hashova za korisničke avatare, **148**

korištenje u funkcijama prikaza, **71-72** korisničkih informacija u, **137** korisnika, hashova lozinke pohranjenih u, **102** korištenje vanjske baze podataka sa Docker kontejneri, **264**

DATABASE_URL varijabla okruženja, **246,** **265**

Validator polja DataRequired, **45, 49, 109** datuma i vremena, datum posljednje posjete za korisnike, **138** lokalizacija sa Flask-Momentom, **38** vremenskih oznaka u bazi podataka o korisnicima, **137**

db objekat, **62**

db.Column klasa, **62**

db.create_all funkcija, **66, 75**

db.ForeignKey funkcija, **64**

db.relationship funkcija, **64** db.session objekat, **67** db.session.add metoda, **68** db.session.commit metoda , **67,** **121** db.session.delete metoda, **68**

db.session.rollback metoda, **68** način otklanjanja grešaka, **13, 93** --debugger i --no-debugger opcije komandne linije, **16** grešaka tokom proizvodnje, **242** podsistema za otklanjanje grešaka, **1** dekorater, **8** prilagođenih, provjeravanje korisničkih dozvola, **132** redoslijed, u prikazu funkcija koje koriste više dekoratora, **133**

zakačice zahtjeva implementirane kao, **20**

DELETE metoda zahtjeva (HTTP), **201** denormalizacija (NoSQL baze podataka), **59** zavisnosti, **1** za aplikacije postavljene na Heroku, **248** za razvoj u odnosu na proizvodnju, **156**

- instaliranje u virtuelno okruženje sa pip-om, 5
- datoteka zahtjeva za velike aplikacije, 93
- komanda raspoređivanja, 241 raspoređivanje, 241-273
- Docker kontejneri, 256-270 u oblaku, 243 greške u evidentiranju tokom proizvodnje, 242 na Heroku platformi, 244-255 priprema aplikacije, 244-253 tradicionalno, 270-273
- uvoz varijabli okruženja, 271 podešavanje servera, 271 podešavanje evidentiranja, 272 radni tok, 241 deserializacija (API), 212 razvojni web server, 10
- DEV_DATABASE_URL varijabla okruženja, 96
- Docker, 244, 256-270
- izgradnja Docker slike, 257 orkestracija kontejnera sa Compose, 265-269
- Dockerfile, 257
- naredba slika, 260, 269 instalacija, 256 naredba za prijavu, 263 naredba dnevnika, 262, 264 ps naredba, 269 push naredba, 263 rm naredba, 262, 269 rmi naredba, 269 naredba za pokretanje, 262, naredba za zaustavljanje
- Roj, 270
- sistemska komanda, 269 naredba oznake, 263 korištenje eksterne baze podataka, 264 korištenje u proizvodnji, 270 naredba verzije, 256 naredba volumena, 269
- Docker Compose, 265
- docker-compose.yml fajl, 266 naredba dnevnika, 268 ps komanda, 269 up komanda, 268 Docker Hub, 263 aplikacije i usluge uključene, 265 dokumentno orijentisane baze podataka, 57
- dinamičke rute, 9
- dinamičkih URL-ova, generiranje sa funkcijom url_for, 37 dynos (Heroku), 244
- ## I
- EC2 usluga (AWS), 243 e-mail, 79 (pogledajte i Flask-Mail)
- konfiguriranje za aplikacije na Heroku, 248 konfiguriranje za Docker kontejner, 261 potvrđivanje korisničkih naloga, 120-124 rukovanje promjenama adresa za korisničke račune, 125
- slanje za greške u aplikaciji, 242
- dijagrami odnosa entiteta, 58 .env
- datoteka, 253, 266-267, 271 varijabla okruženja definirana u Dockerfileu, 258
- uvoz u tradicionalnoj implementaciji, 271 u konfiguraciji velike aplikacije, 87
- Validator polja EqualTo, 116 rukovanje greškama
- API obrađivač grešaka za ValidationError, 212 rukovatelja greškama u nacrtima aplikacije, 91 Flask-HTTPAuth rukovalac greškama, 208 u RESTful web servisima, 204
- sa pregovaranjem sadržaja, 205 stranica o greškama, prilagođeno, 33-36 grešaka, evidentiranje tokom proizvodnje, 242 proširenja direktiva, 30, 32 proširivost Flask-a, xi ekstenzije, 1, 23, 30 dodatnih Flask ekstenzija i paketa,
- 276
- inicijalizacija, 31
- ## F
- tvornička funkcija, kreiranje aplikacija sa, 88 lažnih podataka blog postova, kreiranje, 155-157
- Faker paket, 155 filtera (baza podataka), 27
- offset() filter upita, 157 upita praćenih postova pomoći spajanja baze podataka, 183
- korištenje s upitim baze podataka, 68
- filter_by metoda, 69
- SQLAlchemy filteri za, 69 flash funkcija, 53

Tvornička funkcija aplikacije Flask, 88
 app_errorhandler dekorater, 91, 205 osnovna struktura aplikacije sa više datoteka, 85 prilagođenih naredbi, 96 dinamičkih ruta, 9 ekstenzija, 30

Flask klasa, 7
 flask komandnih opcija, 15
 instaliranja u virtuelno okruženje sa pip-om, 5
 gašenje servera, 230 test klijent, 224-229 rad sa, dodatni resursi za, 275-277

flask db komanda downgrade, 76 flask db migrate komanda, 75-76 flask db stamp komanda, 76 flask db naredba nadogradnje, 75-76 flask deploy komanda, 268 Flask Extension Registry, 276 flask run komanda, 10 --host argument, 16 opcija, 16 flask shell naredba, 15, 66, 72 flask test naredba, --coverage opcija, 222 Flask-Bootstrap, 30-33 bloka, 29 inicijalizacija, 30 instaliranje, 30 brzi makro, 47 wtf.quick_form Jinja2 makro, 111 .quick_form metoda, 47 Flask-HTTPAuth, 206

before_request rukovalac sa autentifikacijom, 208 rukovalac greškama, 208 inicijalizacija, 207 podrška za autentifikaciju zasnovanu na tokenu, 209 Flask -Login, 107-115 dodavanje obrasca za prijavu, 109 AnonymousUserMixin klasa, 132 current_user kontekstualna varijabla, 113, 152 kako radi, 1 Manager_manager_login1 atribut anonymous_user, 132 login_required dekorater, 108, 114 funkcija login_user, 111

funkcija logout_user, 113 priprema korisničkog modela za prijave, 107 prijavljivanja korisnika, 111 korisnika koji se odjavljuju, 112 testiranja prijava, 114

UserMixin klasa, 107 dekorater user_loader, 108

Flask-Mail, 79-84 inicijalizacija, 80 integracija e-pošte sa aplikacijom, 81 slanje asinhronne e-pošte, 83 slanje e-pošte iz Python shell-a, 81 slanje e-pošte preko Gmail-a, 80

Konfiguracijski ključevi SMTP servera, 79

Flask-Migrate, 73-77 dodavanje više migracija, 76 razmatranja u promjeni šema baze podataka, 74 kreiranje ili nadogradnja tablica baze podataka u velikoj aplikaciji, 96 inicijalizacija, 73 nadogradnja baze podataka, 75

Flask-Moment, 38 funkcija formata, 40 funkcija fromNow, 40 funkcija lokalizacije, 41

Flask-PageDown, 162-164 inicijalizacija, 162 Obrazac za objavu s omogućenom uštedom, 162

Flask-SQLAlchemy, 60 metoda dodavanja sesije, 67-68 opcija kolona, 63 metoda create_all, 66, 95 upravljanje bazom podataka sa, 61 metoda brisanja sesije, 68 metoda drop_all, 66 omogućavanje snimanja statistike upita, 239 filter_by filtera upita, 704 first_query_4 metoda, 139 funkcija get_debug_queries, 237 funkcija get_or_404 pogodnosti, 145 modela, 62

MySQL konfiguracija, 61 metoda paginacije, 158 atributa objekta paginacije, 158 metoda objekta paginacije, 159 Postgres konfiguracija, 61 izvršilac upita, 69 filtera upita, 69

- objekt upita (baza podataka), 68
 statistika upita koje je snimio, 238
 upitanih praćenih postova pomoći spajanja baze podataka,
 182
- SQLALCHEMY_DATABASE_URI konfiguracija, 61
- SQLALCHEMY_TRACK_MODIFICATIONS konfiguracija, 61
 SQLite konfiguracija, 61 Flask-
 SSLify, 249 Flask-WTF, 43 klasa
 BooleanField, 109 konfiguracija,
 44 krivotvorene zahtjeve na
 više lokacija (CSRF), 44
 DataRequired validator, 45, 109
 onemogućavanje CSRF tokena u jediničnim
 testovima, 226 Flask4Form klasa forme polja, 45
 Validator dužine, 109 obrazac za prijavu, 109
 klasa PasswordField, 109 renderiranje, 47 klasa
 StringField, 45, 109 klasa SubmitField, 45, 109
 korištenje za renderiranje obrasca, 47
 validate_on_submit metoda, 49, validator Flask
 4, 1116 klasa Flasky, slika Docker kontejnera za,
 260 flasky.py, 266, 272 naredba pokrivenosti, 222
 naredba deploy, 241, 253, 255, 260 naredba
 profila, 239
- FLASKY_ADMIN varijabla okruženja, 83,
 131
- FLASKY_COMMENTS_PER_PAGE varijabla konfiguracije,
 192
- FLASKY_POSTS_PER_PAGE varijabla konfiguracije, 158
- FLASK_APP varijabla okruženja, 10, 66, 93,
 246
 podrazumevano podešavanje, 97
- FLASK_CONFIG varijabla okruženja, 93,
 247
- FLASK_COVERAGE varijabla okruženja,
 223
- FLASK_DEBUG varijabla okruženja, 14, 93
 postavka po defaultu, 97
 pratilaca, 171-187
- relacije baze podataka i, 171-178 na stranici
 profila, 178-180 prikazuje sljedeće postove
 na početnoj stranici, 183-187
- za petlje, 28
 stranih ključeva, 57, 64
 form.hidden_tag element, 47
 form.validate_on_submit metoda, 142 funkcija
 formata, 40 obrazaca (pogledajte web obrasce)
 paket functools, 133
- G**
- g kontekstna varijabla, 18, 21
 GET metoda zahtjeva (HTTP), 19 u
 preusmjeravanju, 51 u RESTful API-
 jima, 201 obrađivač resursa za blog
 postove, 213 funkcija pregleda koja rukuje
 GET zahtjevima, 48 get_flashed_messages funkcija,
 54
- Git**
- preuzimanje primjera koda, 2 servera
 posvećena Heroku aplikaciji, 246 izvornog koda za
 primjere koda, xiii učitavanje aplikacija na Heroku
 server pomoći git push-a, 254
- korištenje s aplikacijama na Heroku, 244
 Gmail, Flask-Mail konfiguracija za, 80
 Gravatar servis, 146
 Gunicorn web server, 251-252
- H**
- heševi
- MDS hash za URL-ove avatara, 146
 keširanje, 148 heširanje lozinke,
 102 korištenje Werkzeug sigurnosnog
 modula, 102
- HEAD metoda zahtjeva (HTTP), 19 pomoć
 od Flask razvojne zajednice, 276
- Heroku platforma, 244-255
 dodavanje Procfile-a, 252
 dodavanje datoteke sa zahtjevima najvišeg nivoa,
 248 addons:naredba kreiranje, 246
 CLI alat, 245
 config naredba, 246
 config:set naredba, 247
 konfiguriranje e-pošte, 248
 konfiguriranje logiranja, 247
 naredba kreiranje, 246

kreiranje Heroku naloga, [245](#) kreiranje aplikacije, [245](#) implementacija nadogradnje aplikacije na, [255](#) implementacija aplikacija na, koristeći git push, [254](#) omogućavanje sigurnog HTTP-a sa Flask-SSLFy, [249](#) naredba za prijavu, [245](#) naredba dnevnika, [247](#) naredba održavanja, [255](#) obezbeđivanje baze podataka, [246](#) pregledavanje dnevnika aplikacija, [255](#) pokretanje proizvodnog web servera, [251](#) testiranje s heroku lokalnom komandom, [253](#) ime hosta (URL-ovi baze podataka), [61](#)
HTML
 Markdown-to-HTML pretvarač, [164](#) prikazivanje obrazaca u, [47-48](#)
 HTTP (bezbedno), omogućavanje sa Flask-SSLFy, [249](#)
 HTTP autentifikacija, [206](#)
 HTTP metode, [19](#) i
 rukovaoci resursima za RESTful web ser-
 vice, [215](#)
 metode zahtjeva u RESTful API-jima, [201](#)
 HTTP statusni kodovi, [21](#) [404](#)
 greška, [33](#), [145](#)
 RESTful API rukovalac greškama za 403 status
 kod, [206](#)
 koji vraćaju RESTful API-ji, [204](#)
 HTTPBasicAuth klasa, [207](#)
 HTTPPie, korištenje za testiranje web usluga, [217-218](#)

I
 I slike (kontejner), [244](#)
 pravljenje slike Docker kontejnera, [257](#) čišćenje, [269](#)

 uvoz datoteka, makroi šablona, [29](#) ukљučenih
 datoteka, [29](#) nasjedivanja u Jinja2 predlošcima,
 [29](#), [31](#) metoda `init_app`, [88](#) metoda `insert_roles`,
 [131](#) integrirano razvojno okruženje (IDE),
 [275](#)

 njegov opasan paket, [119](#)
 generiranje tokena za potvrdu za korisničke račune,
 [119](#) podrška za autentifikaciju zasnovanu na
 tokenima, [209](#)

J
 JavaScript datoteke (Bootstrap), [30](#), [33](#)
 JavaScript, moment.js biblioteka, [38](#) Jinja2
 paket, 1, 26-30 blok direktiva, [29](#) kontrolnih
 struktura, [28](#) za direktivu, [28](#) makro
 direktiva, [29](#)

 direktiva proširenja, [30](#)
 direktiva uvoza, [29](#), [47](#)
 uključuje direktiva, [29](#)
 predložaka za renderiranje,
 [26](#) sigurnih filtera, [28](#)
 postavljenih direktiva, [195](#)
 super makroa, [30](#) varijabli, [27](#)
 filtera za, [27](#) `wtf.quick_form`
 makroa, [110](#) spojeva (baza
 podataka) spojenih za prethodne
 reference, [175](#)

 koristeći u bazi podataka upit praćenih objava,
 [181](#)

 jQuery.js biblioteka, [39](#)
 JSON serijalizirajući
 resurse do i od, 210-213 korištenje u RESTful web
 uslugama, [202](#)
 JSON web potpisi (JWS), [119](#) pomoćnih
 funkcija `jsonify`, [203](#) spojne tablice (pogledajte
 tabele asocijacija (podaci
 baza))

K
 baza podataka ključ/vrijednost, [57](#)
 Kubernetes, [270](#)

L
 kodovi jezika, [41](#) veza,
 [36](#) urednika blog
 postova, [168](#) umjerenih
 komentara u navigacijskoj traci,
 [193](#)

 trajne veze do postova na blogu, 165-167 veza za
 uređivanje profila, [142](#) veza za uređivanje profila
 za administratora, [145](#) do komentara na blog
 postovima, [192](#) do stranice korisničkog profila u
 navigacijskoj traci, [140](#) funkcije lokalizacije, [41](#)

lokalizacija datuma i vremena, 38
konfigurisanje logovanja na Heroku platformi, 247 Docker konfiguracija za, 258 docker logs naredba, 262 greške tokom proizvodnje, 242 pregleda dnevnika aplikacija na Heroku platformi, 255 podešavanje u tradicionalnim implementacijama, 272 funkcija pregleda za prijavu , 111 login_manager.anonymous_user atribut, 132 login_manager.user_loader dekorater, 108 login_required dekorater, 108, 114, 122, 208

M

makroi, 29 MAIL_PASSWORD varijabla okruženja, 80 MAIL_USERNAME varijabla okruženja, 80 make_response funkcija, 21, 185 relacija „više-prema-više”, 65, 172 napredna, 174 pomoćne metode sljedbenika, 176 implementacija kao jedan-prema-više odnosa, 175 samoreferencijalna, 174 više-prema- jedna veza, 65

Markdown, 162-165

konverzija u HTML na serveru, 164 post forma omogućena za, 162 poruke, flešovanje iz obrazaca, argument 53-55 metoda, 48 mikroservisa, 265

Microsoft Windows (pogledajte Windows sistemi) skripte za migraciju (baza podataka), 74

kreiranje pomoću flask db migrate, 75 definicija modela (baza podataka), 62-64 moment.js biblioteka, 38 opcije formatiranja datuma i vremena, 40 MySQL baze podataka, 264-265, 271

N

NameForm klasa, 45 imenskih prostora u nacrtima, 92 NoSQL baze podataka, 57-58 Flask podrška za, xi SQL baze podataka vs., 59

O

OAuth2 autentikacija, 80 objektno-dokumentnih mapera (ODM), 60 objektno-relacijskih mapera (ORM), 60 modela, 62 odnosa jedan prema više, 58, 64

tabela komentara na tabelu korisnika, 189 relacija više-prema-više implementirana kao, 175 upiti, 70 relacija jedan-na-jedan, 65

OPTIONS metoda zahtjeva (HTTP), 19

orkestracija (kontejner) sa dockerom sastaviti, 265-269

P**Biblioteka PageDown**, 162

paginacija velikih kolekcija resursa, 216 dugih lista blog postova, 155-161 dodavanje widgeta za paginaciju, 158-161 kreiranje lažnih podataka na blogu, 155 prikazivanje na stranicama, 157 komentara korisnika na postove, 192

PasswordField klasa, 109

lozinki lozinka u URL-ovima baze podataka, 61 sigurnost, 102-105 ažuriranja i resetiranja za korisničke račune, 125 performansi, 237-240

evidentiranje sporih performansi baze podataka, 237-239 profiliranje izvornog koda, 239-240 dozvola, 127 moderiranja komentara, 193 provjere prilagođenih dekoratera, 132 procjenjivanja za korisnika, 132 u bazi podataka korisničkih uloga, 128 konstanti za dozvole, 128 metoda za upravljanje, 129 uloga podržano s dozvolama, 130 jediničnih testova za, 134 permission_required dekorater, 214 pip, 5

Platforma kao usluga (PaaS), 244

Metoda POST zahtjeva (HTTP), 43 u RESTful API-jima, 201 odgovor preusmjeravanja na POST zahtjeve, 51 rukovalac resursima za objave na blogu, 214 funkcija pregleda koja rukuje POST zahtjevima, 48

- Objavi/Preusmjeri/Nabavi obrazac,
52 prijave i 112
- Postfiks, 271
- Postgres baze podataka, 246, 271
- logika prezentacije, 25 primarni
ključ (baza podataka), 57 kolona
primarnog ključa, Flask-SQLAlchemy
zahtjev za, 64
- Profil profila (aplikacije na Herokuu), 252 CSS
klasa zaglavlja profila, 147 CSS klasa sa sličicama
profila, 147 profila, 137-149 korisničko ime i
avatar autora bloga, linkovi na stranicu profila,
153 objave na blogu na stranicama profila, 154
kreiranje korisničkog profila strana, 138-141
urednik za, 141-145
- urednik na nivou administratora, urednik
na nivou korisnika 143-145, 141 pratilac
na stranici profila, 178-180 informacija u, 137
korisničkih avatara, 146-149 izvorni kod za
profilisanje, 239 proxy servera, 250, 271
psycopg2 paket, 248
- PUT metoda zahtjeva (HTTP), 201
PUT obrađivač resursa za blog postove, 215
pymysql paket, 264
Python, 1
kreiranje virtuelnih okruženja sa Pythonom
2, 3
kreiranje virtuelnih okruženja sa Pythonom
3, 3
okvira baze podataka, 59-60
integracija baze podataka sa Python školjkom, 72
instalacija paketa sa pip-om, 5 slika interpretatora
na Docker Hub-u, 258
Indeks Python paketa, 276
python-dotenv paket, 272
- Q
objekt upita (baza podataka), 68
- R
funkcija preusmjeravanja, 22, 53
preusmjeravanja, 22, 51-53
SSL, 249
- Validator polja obrasca Regexp, 116
registrovani novih korisnika, 115-118
- dodavanje obrasca za registraciju korisnika,
115 registrovanih korisnika, 117 regresija,
221 relaciona baza podataka, 57
- (vidi također SQL baze podataka)
- Flask podrška za, xi
- relacije (baza podataka), 57, 64-65, 171-178 dinamičke
veze, 70 mnogo-prema-više, 172 napredno, 174
ispitivanje odnosa jedan-prema-više, 70
samoreferencijalno, 174
- Opcije SQLAlchemy za, 65
relativni URL-ovi (u linkovima), 37
REMEMBER_COOKIE_DURATION opcija,
111
- protokoli poziva udaljene procedure (RPC), 199
renderiranje (šabloni), 25 funkcija render_template,
27, 49, 53
- Kontekst zahtjeva za prijenos stanja (vidi REST), 18
ciklus zahtjev-odgovor, 17-22 konteksti aplikacije i
zahtjeva, 17
- Metode HTTP zahtjeva, 19 tijela
zahtjeva i odgovora u RESTful-u
API-ji, 201
slanje zahtjeva, 18 zakačivačih
zahtjeva, 20 metoda zahtjeva
u RESTful API-jima, 201 objekt zahtjeva, 19
zahtjeva, 8 formata odgovora za RESTful API
klijente,
- 205
- odgovori, 8, 21
objekt odgovora, 21
- datoteka sa zahtjevima, 93, 248
datoteka sa zahtjevima za razvoj, 156
datoteka sa zahtjevima najvišeg nivoa za Heroku
platformu, 248
- resurse
- implementacija krajnjih tačaka za, 213-216
paginaciju velikih kolekcija, 216 serijaliziranje
u i iz JSON-a, 210-213
URL-ovi za, 202
- resursa za rad sa Flask, 275-277 dobijanje pomoći,
276 uključivanje u Flask zajednicu,
- 277

integrisana razvojna okruženja
(IDE), 275

REST, 199

- definiranje karakteristika za arhitekturu web servisa, 199 tijela zahtjeva i odgovora, 201
- metoda zahtjeva, 201 resurs, koncept, 200
- verzija web servisa, 202

RESTful web usluge sa Flaskom, 203-218

- kreiranje API nacrtu, 203 rukovanje greškama, 204 implementacija krajnjih tačaka resursa, 213-216 paginacija velikih kolekcija resursa, 216 serijalizacija resursa u i iz JSON-a, 210-213

testiranje sa HTTPPie, 217-218

autentikacija zasnovana na tokenu, 208-210 autentikacija korisnika sa Flask-HTTPAuth, 206

obrnuti proxy serveri, 250, 271

bogata internet aplikacija (RIA), 199 postova obogaćenog teksta, koristeći Markdown i Flask PageDown, 161-165

- rukovanje bogatim tekstom na serveru, 164 uloge, 127-135 dodavanje razvojnoj bazi podataka u sesiji ljske, 134 dodjela, 131 predstavljanje baze podataka, 127-131 uloga kreiranja metoda, 130 jediničnih testova za, 134 verifikacija, 132-135 vraćanje sesija baze podataka, 68

Ronacher, Armin, 1

dekorater rute, 133

rute, 8

- pristup samo provjerjenim korisnicima, 108 generiranje tokena za autentifikaciju, 210 podrška za komentare na blogu, 191 moderiranje komentara, 196 dinamičkih, 12 uređivača za blog postove, 167 praćenje rute, 179 ruta početne stranice s objavama na blogu, 152 podrška za paginaciju, 157 u nacrtu autentifikacije, 105 u nacrtima, 91 stalni link do postova na blogu, 165

ruta za uređivanje profila, 141 ruta za uređivanje profila za administratore, 144 ruta stranice profila, 138 ruta stranice profila sa objavama na blogu, 154 ruta za registraciju s potvrdom e-pošte, 120

izbor svih ili praćenih objava, 184 odjava, 113 registracija korisnika, 117 podsistema rutiranja, 1 red (baza podataka), 57

brisanje, 68 umetanje, 66 modifikacija, 68 upiti, 68

S

tajni ključ, 44, 87, 119

- za aplikacije na Heroku platformi, 247 siguran HTTP, 249

SelectField class, 144

Selen, end-to-end testiranje sa, 230-234

samoreferencijalnih odnosa, 174

Sendmail, 271

serijalizacija

- deserializacija resursa iz JSON-a, 212
- serijalizacija resursa u JSON, 210 podešavanje servera u tradicionalnoj implementaciji, 271

gašenje servera, 230 varijabla konteksta sesije, 18, 52 session.get metoda, 53 sesije (baza podataka), 67 urezivanja, 67 vraćanja, 68 sesija (korisnik), 44 preusmjeravanja i, 51 set_cookie metoda, 21, 185 procesor konteksta ljske, 72 SMTP server, 79 smtplib paket, 79

profiliranje izvornog koda, 239 SQL (Structured Query Language), 57 SQL baza podataka, 57

NoSQL vs., SQLAlchemy, 60 opcija stupaca, 63

tipa kolona, 63 inspekcija izvornog SQL upita generiranog od strane,

Markdown konverzija teksta u HTML, [164](#) izvršitelja upita, [69](#) filtera upita, [69](#) opcija odnosa, [65](#)

`SQLALCHEMY_DATABASE_URI`, [61](#), [87](#)
`SQLALCHEMY_TRACK_MODIFICATIONS`, [61](#)

SQLite baze podataka, [271](#)
`SSL_REDIRECT` varijabla, [249](#), [250](#) tragova steka, [242](#) web servisa bez stanja, [206](#) statičkih datoteka, [37](#) statičkih metoda, [131](#) statusni kod (HTTP), [21](#)

`StringField` klasa, [45](#), [109](#)
`SubmitField` klasa, [45](#), [109](#) super funkcija, [30](#), [33](#) syslog, [272](#)

T
`_tablename_` varijabla klase, [62](#) tabele (baza podataka), [57](#) kreiranje, [66](#) kreiranje ili nadogradnja u velikoj aplikaciji, [96](#)
spaja, [58](#)
`teardown_request` kuka, [20](#) šablona, [25](#)-[41](#) dodavanje klase dozvole u kontekst šablona, [134](#)
Bootstrap integracija koristeći Flash Bootstrap, [30](#)-[33](#)
moderiranja komentara, [194](#) e-poruka potvrde koje koristi ncrt autentifikacije, [121](#) prilagođena stranica o grešci, [33](#)-[36](#) definirano, [25](#) uređivanje šablona blog postova, [167](#) renderiranja flash poruke, [54](#)

Deklaracija predloška Flask-PageDown, [162](#) poboljšanja pratioča u zaglavju profila, [178](#)
za ncrt autentifikacije, [105](#) za poruke e-pošte, [81](#) za formular za prijavu, [109](#) za obrazac za registraciju novog korisnika, [116](#) za korisnički profil, [139](#) pozdrav prijavljenog korisnika, [114](#) predložak za početnu stranicu sa objavama na blogu, [152](#)

Jinja2 šablonski mehanizam, [26](#)-[30](#) linkova, [36](#) lokalizacija datuma i vremena sa Flaskom
Trenutak, [38](#)
predložak za prijavu, ažuriranje radi prikazivanja obrasca za prijavu, [112](#) paginacija podnožja za liste postova na blogu, [160](#) makro predložaka paginacije, [159](#) trajnih veza do postova na blogu, [166](#) predložaka stranice profila sa objavama na blogu, [155](#) statičnih datoteka, [37](#) foldera sa šablonima, [26](#) korištenje da prikažete obrazac u HTML-u, [47](#)

test komanda za pokretanje jediničnih testova, [95](#) testiranje, [221](#)-[235](#) procena vrednosti, [234](#) end-to-end, korišćenje Selena, [230](#)-[234](#) dobijanje izveštaja o pokrivenosti koda, [221](#)-[224](#) web servisa sa HTTPie-om, [217](#)-[218](#) testovi heširanja lozinka, [104](#) fajla jediničnih testova za velike aplikacije, [94](#) jediničnih testa za uloge i dozvole, [134](#) koristeći Flask test klijent, [224](#)-[229](#)

testiranje web aplikacija, [225](#)-[228](#) testiranje web servisa, [228](#)-[229](#) provjera funkcionalnosti prijave, [114](#) vremenskih (vidi datume i vrijeme) vremenskih oznaka, rad sa, korišćenjem Flask-a Trenutak, [39](#)
autentifikacija zasnovana na tokenima, [208](#)-[210](#), [218](#) transakcija, [67](#) (vidi također sesije (baza podataka))

IN
nepotvrđeni računi filtriranje u obrađivaču `before_app_request`, [122](#) stranica na kojoj se traži potvrda naloga, [123](#) unittest paket, [95](#)

URL fragmenti, [192](#)
URL-ovi rute aplikacija, [9](#) URL mapa aplikacije, [19](#) avatar, [146](#) baza podataka, u Flask-SQLAlchemy, [61](#) za resurse, [200](#) za resurse potpuno kvalifikovan, [202](#)

u e-porukama potvrde za korisničke račune,
121
u vezama, 37
funkcija url_for, 37, 53, 92, 121
argument url_prefix, 106 autentikacija
korisnika (vidi autentifikaciju) komentari korisnika,
189-197 predstavljanje baze podataka, 189-191
moderiranje, 193-197 podnošenje i prikaz-,
193 funkcija učitavanja korisnika, 108

Model korisnika

priprema za prijavu, 107
priprema za heširanje lozinke, 102 generisanje
tokena korisničkog naloga i verifika-
cija, 119
korisničkih profila (pogledajte profile)
korisničke uloge (pogledajte uloge)
korisničke sesije, 44, 52
isteka, dugoročni kolačić za, 111 User.can
metoda, 153 korisničko ime (URL-ovi baze
podataka), 61
korisnika
čineći postojeće korisnike svojim sljedbenicima,
186
stvaranje vlastitih sljedbenika na izgradnji, 186

UTC (koordinirano univerzalno vrijeme), 38
uWSGI web server, 252

V

metod validate_on_submit, 49, 111
ValidationError, 116, 212
RESTful API rukovalac greškama, 213
validadora, 44, 109 ugrađenih, WTForms paket,
46 implementiranih kao metode, 116
varijabli, 26-27 filtera za, 27 venv paket
(Python), 3 verify_password metoda, 111
funkcija pregleda, 8 potvrda korisnički
računi, 121 korištenje baze podataka, 71-72 za
komentare na blog postovima, 191 za uređivač
postova na blogu, 168 za pratioce na stranici
profila, 179 za trajne veze do postova na
blogu, 165 za obradu obrasca, 48-51

u autentifikacijskom nacrtu, 105 u
nacrtima, 92 implementacije funkcije
za prijavu, 111 redoslijed dekoratora, 133
svrhe, 25 virtualnih okruženja, 2 aktiviranje,
4 kreiranje s Pythonom 2, 3 kreiranje s
Pythonom 3, 3 deaktiviranje, 5 instaliranje Flask
u, 5 korištenje bez aktivacije, 5 virtuelnih
mašina, 256 virtuelnih servera, 243
virtualenv uslužni program, 3 volumena
(Docker), uklanjanje, 269

U

Web server za konobarice,
252 web pretraživača, alat za automatizaciju Selenium,
230 web dynos (Heroku), 244 web formulara, 43-55
obrasca za blogove, 151 konfiguracija, 44 treperenje
poruke, 53-55 klase obrazaca, 44-47 generirane od
strane Flasha -WTF, CSRF tokeni u,

226

rukovanje funkcijama prikaza, 48-51
HTML prikaz, 47-48 u nacrtima,
92 formular za prijavu, 109
obrazac za uređivanje profila ,
141 obrazac za uređivanje profila
za administratore, 143 preusmjeravanja i korisničke
sesije, 51-53 obrazac za registraciju korisnika, 115

Web Server Gateway Interface (WSGI), 1, 7 web
servera koji se instaliraju u tradicionalnoj

implementaciji, 271 pokrenuti proizvodni web
server na Heroku,
251

web servisi, 199, 203

(pogledajte i RESTful web usluge sa Flaskom)
testiranje RESTful web servisa koristeći Flask test klijent,
228-229

Alat (sigurnosni modul), 1, 102, 242
generate_password_hash funkcija, 103
ProfilerMiddleware WSGI srednji softver, 240
ProxyFix WSGI srednji softver, 250

metoda verify_password, 103
Windows podsistem za Linux (WSL), 2
Windows sistemi, 2
Docker za Windows, Hyper-V funkcija i,
257
testiranje Heroku implementacije, korištenjem
Waitress web servera, 252 radni dynos
(Heroku), 244 wtf.quick_form funkcija, 47

WTForms paket, 43
ugrađeni validatori, 46
Validator regularnih izraza, 116
SelectField klasa omotača, 144
standardna HTML polja podržana od, 45

X

XML u RESTful web servisima, 202

O autoru

Miguel Grinberg ima preko 25 godina iskustva kao softverski inženjer. Ima [blog](#) gdje piše o raznim temama, uključujući web razvoj, robotiku, fotografiju i povremene filmske pregledе. Živi u Portlandu, Oregon.

Kolofon

Životinja na naslovnoj strani Flask Web Development-a je pirenejski mastif (pasmina *Canis lupus familiaris*). Ovi divovski španski psi potječe od drevnog psa čuvara stoke zvanog Molossus, kojeg su uzgajali Grci i Rimljani i koji je sada izumro. Međutim, poznato je da je ovaj predak igrao ulogu u stvaranju mnogih rasa koje su danas uobičajene, kao što su rotvajler, nemačka doga, njufaundlend i kane korso. Pirenejski mastifi su priznati kao čista pasmina tek od 1977. godine, a Američki klub pirenejskih mastifa radi na promociji ovih pasa kao kućnih ljubimaca u Sjedinjenim Državama.

Nakon Španjolskog građanskog rata, populacija pirenejskih mastifa u njihovoј domovini je opala, a pasmina je opstala samo zahvaljujući predanom radu nekoliko raštrkanih uzgajivača diljem zemlje. Moderni genetski fond za Pirinejce potiče iz ove poslijeratne populacije, što ih čini sklonim genetskim bolestima poput displazije kuka. Danas, odgovorni vlasnici se pobrinu da njihovi psi budu testirani na bolesti i rendgenski snimljeni kako bi se potražile abnormalnosti kuka prije nego što se uzgajaju.

Odrasli mužjaci pirenejskog mastifa mogu doseći i više od 200 funti kada su potpuno odrasli, tako da posjedovanje ovog psa zahtijeva posvećenost dobroj obuci i puno vremena na otvorenom. Uprkos svojoj veličini i istoriji lovaca na medvjede i vukove, pirenejci imaju vrlo miran temperament i odličan su porodični pas. Na njih se može osloniti da će brinuti o djeci i zaštititi dom, dok su u isto vrijeme poslušni s drugim psima. Uz odgovarajuću socijalizaciju i snažno vodstvo, pirenejski mastif uspijeva u kućnom okruženju i bit će odličan čuvar i pratilac.

Mnoge životinje na naslovnicama O'Reillyja su ugrožene; svi oni su važni za svijet. Da sazname više o tome kako možete pomoći, idite na animals.oreilly.com.

Naslovna slika je iz JG Wood's Animate Creation. Fontovi naslovnice su URW Typewriter i Guardian Sans. Font teksta je Adobe Minion Pro; font naslova je Adobe Myriad Condensed; a kodni font je Ubuntu Mono Daltona Maaga.