Razvoj web-aplikacija pomoću aplikacijskih okvira

Penezić, Matea

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matea Penezić

RAZVOJ WEB-APLIKACIJA POMOĆU APLIKACIJSKIH OKVIRA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, rujan, 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Hvala mami, tati, sestri i Borisu koji su mi bili velika podrška tijekom čitavog studiranja.

Sadržaj

Uvod

Prve web aplikacije počele su se razvijati 90-ih godina prošlog stoljeća. Popularnost interneta uzrokovala je veću potražnju za web aplikacijama, te kako se unapređivalo sklopovlje osobnih računala, tako je rasla i kompleksnost web aplikacija. Većina web aplikacija dijele neke zajedničke funkcionalnosti, npr. komunikacija s bazom podataka, upravljanje sesijom, ispis HTML-a itd. Kako programeri ne bi sa svakim novim projektom ponovo razvijali navedene funkcionalnosti, njihove apstrakcije su izdvojene u biblioteke. Kolekcija biblioteka koje zajedno čine jednu cjelinu naziva se aplikacijskim okvirom.

Aplikacijski okviri ubrzavaju proces izrade web aplikacija. Danas je u većini programskih jezika napisan barem jedan aplikacijski okvir za razvoj web aplikacija. Jedan od njih je Django te su njegove značajke opisane u ovom radu.

Rad je podijeljen u sedam poglavlja. Prvo poglavlje daje kratki uvod u aplikacijske okvire te nakon toga slijede upute o instalaciji Djanga i pregled strukture Django projekta i Django aplikacije. Na kraju poglavlja opisana je arhitektura Djanga. U drugom poglavlju opisano je kako se kreiraju modeli, njihovi atributi i relacije te kako se kreiraju tablice u bazi podataka koje odgovaraju modelima. Treće je poglavlje posvećeno pogledima, kako se definiraju te povezuju s modelima. Slijedi poglavlje o predlošcima, Djangovom jeziku predložaka i povezivanju predložaka s pogledima. U petom se poglavlju opisuju Django obrasci te kako ih je moguće definirati na temelju postojećih modela. Šesto poglavlje daje osnovne informacije o Django autentifikacijskom sustavu te se opisuje `User` model i njegove značajke. Posljednje poglavlje vezano je uz Django web aplikaciju napravljenu u sklopu ovog rada, definirane su aplikacije unutar projekta, modeli, te su opisane funckionalnosti cijele aplikacije.

Poglavlje 1

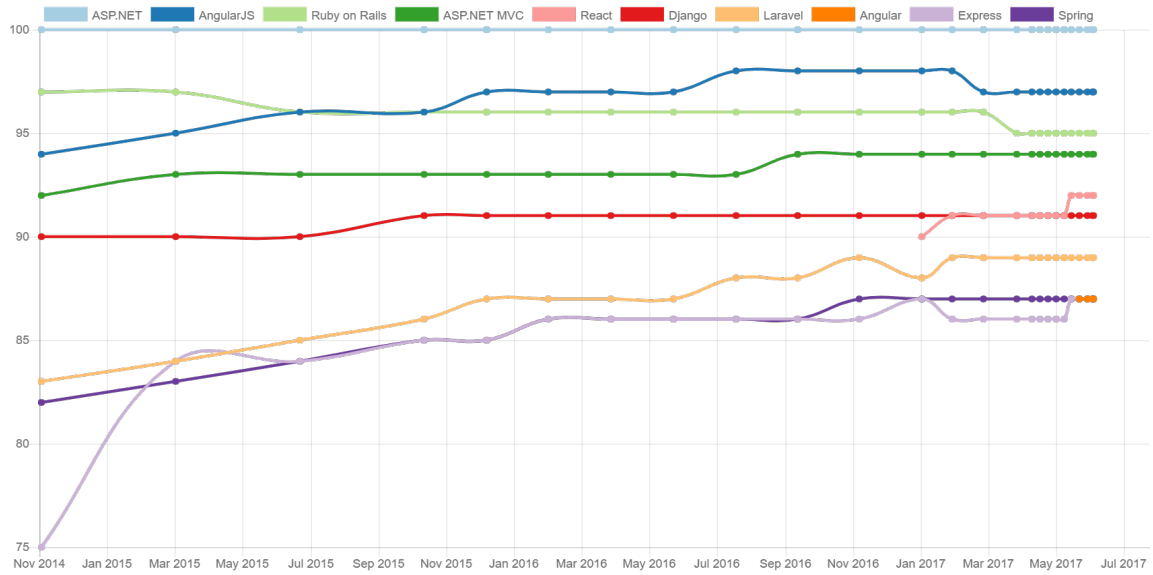
Aplikacijski okviri i Django

1.1 Web aplikacijski okvir

Web aplikacijski okviri (eng. *web framework*) su programski okviri koji olakšavaju razvoj web aplikacija uključujući web servisa i web API-a. Pružaju biblioteke za jednostavniji pristup i komunikaciju s bazom, upravljanje sesijama, pogon za predloške (eng. *templating engine*) te promoviraju ponovnu uporabu napisanog koda.

Najčešće se koriste kod dinamičkih web stranica, ali ih je moguće koristiti i za statičke. Većina web aplikacijskih okvira temelji se na MVC arhitekturi koja razdvaja podatkovni model s poslovnom logikom od korisničkog sučelja.

Aplikacijski okviri mogu se podijeliti na klijentske i poslužiteljske. Neki od klijentskih okvira su Backbone.js (JavaScript), AngularJS (JavaScript), EmberJS (JavaScript), ReactJS (JavaScript) i Vue.js (JavaScript), a od poslužiteljskih Django (Python), Spring (Java), Ruby on Rails (Ruby) i Zend Framework (PHP). Na slici 1.1 prikazano je kako se popularnost najpopularnijih aplikacijskih okvira kretala od kraja 2014. godine. Popularnost pojedinog okvira mjeri se GitHub rezultatom koji se temelji na broju dobivenih zvjezdica i Stack Overflow rezultatom koji ovisi o broju postavljenih pitanja koja imaju oznaku pojedinog aplikacijskog okvira. Ukupan rezultat normaliziran je na raspon 0 - 100.



Slika 1.1: Najpopularniji web aplikacijski okviri (`https://hotframeworks.com/`)

1.2 Uvod u Django

Django je aplikacijski okvir koji služi za razvoj web aplikacija. Napisan je u Python programskom jeziku te je besplatan i otvorenog koda. Slijedi MTV (model-template-view) arhitekturalni uzorak. Jedna od velikih prednosti Djanga je što olakšava i ubrzava izradu kompleksnih web-aplikacija koje koriste bazu podataka za pohranu informacija te dohvaćanje i prikazivanje istih prilikom učitavanja stranice (eng. *database-driven website*).

Neke od poznatih stranica koje koriste Django su: Pinterest, Instagram, Mozzila, Bitbucket, Public Broadcasting Service.

Nastao je 2003. godine kada su programeri Adrian Holovaty i Simon Willison, iz Lawrence Journal-World-a počeli koristiti Python za izradu aplikacija. U javnost je pušten u srpnju 2005.g. Dobio je ime prema gitaristu Django Reinhardt-u.

Od 2008.g. održavanje je preuzela Django Software Foundation (DSF) neprofitna organizacija.

1.3 Instalacija Djanga

Za uspješnu instalaciju Djanga potrebno je proći kroz sljedeća tri koraka:

1. instalirati Python i Pip

   Prvo je potrebno instalirati Python, pošto je Django Python aplikacijski okvir. Skripta za instalaciju Pythona može se pronaći na stranici: `http://www.python.org/downloads/`. Tijekom pisanja ovog rada korištena je verzija 3.4.4. Prilikom instalacije korisno je dodati Python u varijable okruženja označavanjem odgovarajućeg potvrdnog okvira (eng. *checkbox*), iako se to može napraviti i naknadno. Na taj se način prilikom korištenja Pythona ne treba pisati cijela putanja do skripte već samo `python`. Pip je alat koji služi za upravljanje softverskim paketima napisanim u Pythonu te omogućava jednostavnu instalaciju i deinstalaciju Python paketa. Upute za instalaciju Pip-a mogu se naći na stranici: `http://www.pip-installer.org/`.

2. instalirati virtualno Python okruženje

   Drugi korak nije nužan, ali je koristan i sprječava konflikte koji mogu nastati između projekata na računalu koji koriste različite verzije Djanga, Pythona te biblioteka unutar pojedinog projekta. Virtualno Python okruženje omogućava da se svaki projekt kreira unutar izoliranog okruženja te na taj način ne utječe ni na šta izvan njega. Jedno od takvih okruženja je Virtualenv. Može se instalirati pomoću naredbe: `pip install virtualenv`. Sada se može kreirati novo virtualno okruženje pomoću: `virtualenv ime_virtualnog_okruženja`.

   Prilikom kreiranja novog virtualnog okruženja, automatski će se instalirati i Python unutar njega. Ukoliko na računalu postoji više verzija Pythona, a ne želi se koristiti zadana verzija Pythona moguće je instalirati neku od drugih verzija pomoću `virtualenv -p putanja_do_pythona ime_virtualnog_okruženja`. Nakon kreiranja novog okruženja, ono se automatski aktivira, a za svaku sljedeću upotrebu potrebno je unutar direktorija ime_virtualnog_okruženja pokrenuti naredbu: `Scripts\activate`. Za deaktivaciju se koristi naredba: `Scripts\deactivate`.

3. instalirati Django

   Konačno se može instalirati Django unutar aktiviranog virtualnog okruženja: `pip install django`.

   Prethodna naredba instalirat će zadnju verziju Djanga. Ukoliko se želi instalirati neka od prethodnih, umjesto 'django' se upiše npr. `django==1.7`.

Napisane naredbe radit će ako su Pip i Virtualenv varijable okruženja, inače je potrebno pisati cijelu putanju.

Opisan proces instalacije vrijedi za Windows operativni sustav.

1.4 Django projekt

Za kreiranje novog projekta potrebno je aktivirati virtualno okruženje koje se želi koristiti. Otvori se direktorij unutar kojeg se želi kreirati projekt te upiše naredba (ime projekta ovog rada je licensing): `django-admin startproject licensing`. Unutar novog projekta nalazi se sljedeće:

```
licensing/
    manage.py
    licensing/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Vanjski `licensing` direktorij predstavlja spremište za projekt, za Django nije bitno kako se zove ovaj direktorij. `manage.py` je skripta koja omogućava interakciju s projektom putem naredbenog retka. Unutarnji `licensing` direktorij je Python paket za kreirani projekt. `__init__.py` je prazna datoteka čije ime daje do znanja Pythonu da je direktorij u kojem se ona nalazi Python paket.

Unutar `settings.py` definirana je konfiguracija za Django projekt. Django mora znati koje postavke se koriste tako da je potrebno definirati varijablu okruženja `DJANGO_SETTINGS_MODULE`. Njena vrijednost mora biti na Python putanji (npr. `licensing.settings`). Postoje i zadane vrijednosti svih postavki, tj. `settings.py` datoteka može biti prazna ukoliko nam odgovaraju zadane vrijednosti. Zadane vrijednosti svih postavki definirane su unutar datoteke `django/conf/global_settings.py`. Naredbom `python manage.py diffsettings` može se provjeriti koje su sve postavke definirane, a da su različite od zadanih. U Django aplikacijama postavke se koriste na način da se uključi objekt `django.conf.settings`. Npr.

```
from django.conf import settings

if settings.DEBUG:
    #...
```

Postavke je moguće koristiti na još jedan način, bez postavljanja vrijednosti varijable okruženja `DJANGO_SETTINGS_MODULE`. Tada se unutar datoteke moraju definirati postavke koje se žele koristiti pomoću funkcije `django.conf.settings.configure(default_settings, **settings)`, npr.

```
from django.conf import settings

settings.configure(TIME_ZONE = 'UTC', LANGUAGE_CODE = 'en-us')
```

Ako se Django komponente koriste samostalno (kao što će se koristiti u primjeru s populacijskom skriptom 2.3), potrebno je nakon postavljanja vrijednosti `DJANGO_SETTINGS_MODULE` ili poziva funkcije `configure()` pozvati funkciju `django.setup()`.

U `urls.py` skripti deklariraju se URL adrese koje se koriste u projektu. Odmah nakon kreiranja projekta u `urls.py` skripti nalazi se sljedeće:

```
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

Na URL adresi `http://127.0.0.1:8000/admin/` nalazi se sučelje za administraciju kojemu mogu pristupiti samo superkorisnici. Ukoliko se taj URL izbriše iz liste `urlpatterns` sučelje za administraciju neće biti dostupno.

Ostala je još `wsgi.py` datoteka koja predstavlja ulaznu točka za WSGI[1]-kompatibilne web poslužitelje te se koristi u produkciji. U njoj se npr. definira okolina za produkciju, odnosno postavke koje se koriste:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'licensing.settings'
```

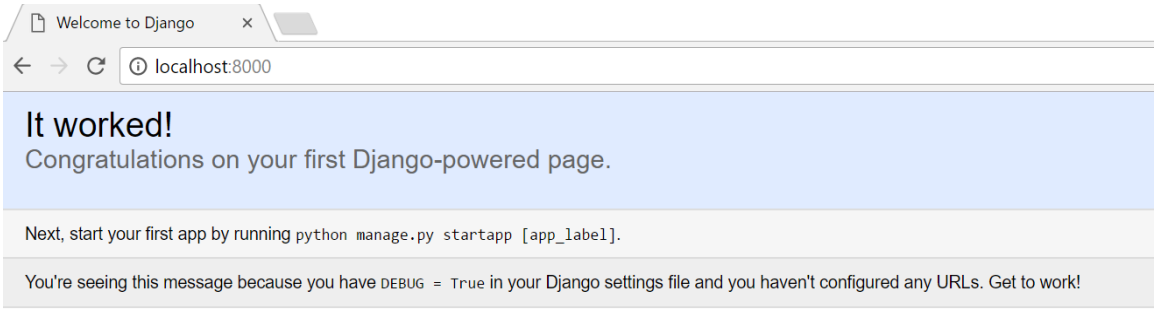
Pokretanje poslužitelja

Naredbom `python manage.py runserver` pokreće se Django poslužitelj na localhost-u koji sluša na portu 8000.

```
Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migrations. Your project may not work properly until you
apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 13, 2017 - 10:14:08
Django version 1.11, using settings 'licensing.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Ukoliko je projekt uspješno kreiran, otvaranjem preglednika na URL adresi `http://127.0.0.1:8000/` vidi se stranica prikazana na slici 1.4:

[1] WSGI je specifikacija za standardizirano sučelje između web poslužitelja i Python web aplikacija.



Slika 1.2: Welcome to Django

Na prethodnom isječku može se vidjeti rečenica u kojoj piše kako postoje migracije koje nisu izvršene. Migracije su naziv za funkcionalnost koja olakšava upravljanje bazom podataka, odnosno njezinom strukturom, ovisno o izgledu modela aplikacije. Na taj način dobije se sustav za verzioniranje baze podataka. One npr. omogućavaju jednostavnije vraćanje na neko od prethodnih stanja modela ili u slučaju kada više programera radi na istom projektu, svi imaju na uvid napravljene promjene na modelima koje lokalno mogu izvršiti na bazi jednom naredbom. Migracije se izvršavaju naredbom: `python manage.py migrate`. Naredba `migrate` prikuplja sve migracije koje još nisu izvršene (postoji `django_migrations` tablica u bazi u kojoj su pohranjene sve izvršene migracije) te ih pokreće.

1.5 Django aplikacija

Django projekt sastoji se od više Django aplikacija tj. manjih cjelina koje obavljaju određenu funkcionalnost. Aplikacije se mogu uključiti i unutar drugog Django projekta, odnosno mogu se ponovo upotrijebiti što znači da je moguće unutar svog Django projekta koristiti i aplikacije drugih ljudi. Nova aplikacija kreira se unutar projekta ovako: `python manage.py startapp products`.

Unutar nove aplikacije nalazi se sljedeće:

```
companies/
    migrations/
        __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

`migrations` je direktorij unutar kojeg se pohranjuju migracije na bazi podataka. `admin.py` je datoteka unutar koje je moguće registrirati modele što će omogućiti upravljanje tim modelima preko admin sučelja. `apps.py` je skripta za definiciju konfiguracije specifične za aplikaciju. Unutar `models.py` definiraju se modeli aplikacije. `tests.py` je datoteka za pohranu funkcija za testiranje. U `views.py` datoteci definiraju se funkcije koje primaju pozive na server i vraćaju odgovore.

Nakon kreiranja nove aplikacije potrebno ju je dodati unutar INSTALLED_APPS liste koja se nalazi u settings.py. Kao što se može vidjeti, Django ima i neke ugrađene aplikacije.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'companies',
]
```

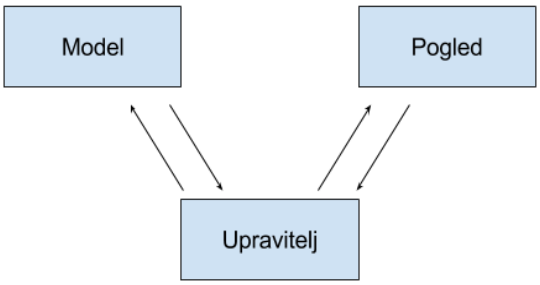
1.6 MVC i MVT arhitektura

MVC

Oblikovni obrasci su generička rješenja za učestale probleme u određenom kontekstu u softverskom razvoju. Postoje razni oblikovni obrasci koji rješavaju različite vrste problema. Jedan od poznatih arhitekturalnih oblikovnih obrazaca je Model-Pogled-Upravitelj (eng. *Model-View-Controller*) odnosno MVC, koji raspodjeljuje aplikaciju na tri komponente određene namjene koje međusobno komuniciraju. Model sadrži poslovnu logiku te pruža sučelje za dohvat i spremanje podataka. Komunicira sa spremištem podataka (bazom podataka, vanjskim servisima i sl.). Upravitelj (eng. *Controller*) odgovoran je za upravljanje korisničkim zahtjevima i unosima. Pogled (eng. *View*) prikazuje tražene podatke u određenom formatu.

Sama interakcija između komponenti definirana je sljedećim pravilima (1.6):

- Upravitelj obrađuje korisnički zahtjev, komunicira s modelom te proslijeđuje podatke komponenti pogled.
- Model sprema ili dohvaća podatke ovisno o zahtjevima iz upravitelja.
- Pogled generira izlaz (npr. HTML) koji će biti proslijeđen i prikazan korisniku.



Slika 1.3: MVC arhitektura

Velika prednost MVC arhitekture je zasebna enkapsulacija poslovne logike i prezentacijskog sloja, što kao posljedicu ima čitljiviji kod, lakše održavanje te mogućnost rada više programera na različitih slojevima aplikacije bez konflikata prilikom implementacije.

S druge strane, nedostataci su potencijalno kompleksna struktura aplikacije budući da je cilj preraspodjela aplikacije na više slojeva. Slojevitost aplikacije utječe i na krivulju učenja, odnosno programerima koji nisu upoznati s takvom arhitekturom treba više vremena da budu produktivni.

MVC je prvi puta implementiran u Smalltalk programskom jeziku te se počinje primjenjivati u razvoju desktop aplikacija te kasnije web aplikacija. Trenutačno se koristi u velikom broju poznatih aplikacijskih okvira: Ruby on Rails, ASP.NET MVC, Codeigniter, Zend Framework, AngularJS.

MTV

Django je modeliran oko MVC okvira te svoju arhitekturu definira kao MTV (eng. *Model-Template-View*) arihtektura, pritom zamjenjujući komponentu upravitelj s komponentom pogled, te komponentu pogled s komponentom predložak (eng. *template*). Razlog takvoj terminologiji je drugačija interpretacija MVC dizajn uzorka.

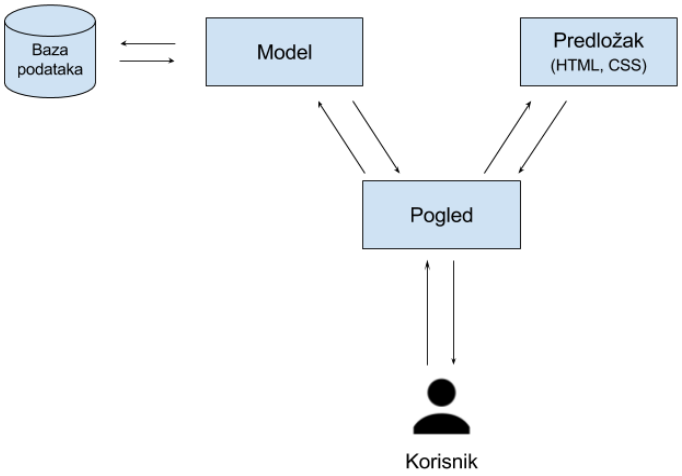
Određene funkcionalnosti upravitelja su ugrađene u Django framework te kako postoje različita mišljenja za što je svaka od komponenti odgovorna, autori Djanga su odlučili koristiti drugačije termine kako bi izbjegli debate oko pravilne implemetacije MVC arhitekture.

Primjer povezivanja pogleda, modela i predloška kako bi se korisniku poslužila stranica nakon što je napravio zahtjev prema poslužitelju je sljedeći (1.6):

1. U `views.py` datoteci se uvezu svi modeli koji će se koristiti.

2. Pogled izvršava upit prema modelu kako bi se spremili ili dohvatili podaci.

3. Rezultat od modela se prosljeđuje predlošku.
4. Predložak se uredi kako bi mogao prihvatiti i prikazati podatke od modela.
5. Mapira se URL s pogledom.



Slika 1.4: MTV - povezivanje komponenti

Poglavlje 2

Modeli

2.1 Sustav za upravljanje bazom podataka

Uobičajena baza podataka za Django je SQLite. SQLite je uključen u Django tako da za njegovo korištenje nije potrebno instalirati ništa dodatno. Međutim, za projekt koji ide u produkciju i koji će imati puno korisnika, poželjno je koristiti neku od drugih baza za koju Django ima podršku (pošto SQLite ne omogućava istovremeno pisanje više korisnika u bazu). Dvije baze podataka koje se najčešće koriste s Djangom su PostgreSQL i MySQL, ali ima podršku i za Oracle bazu. U ovom projektu korištena je MySQL.

Django ima podršku za MySQL 5.5. na više. Za MySQL je potrebno instalirati jedan od tri MySQL DB API upravitelja (eng. *driver*) - MySQLdb, mysqlclient ili MySQL Connector/Python. MySQLdb jedini nema podršku za Python 3. U ovom projektu koristi se mysqlclient koji se može instalirati pomoću: `pip install mysqlclient`.

Za kombinaciju operativnog sustava, verzije Pyhona i Djanga koja je korištena u projektu nije bilo moguće instalirati najnoviju verziju (bez instalacije nekih dodatnih paketa) pa je instalirana verzija 1.3.4.

Unutar `settings.py` datoteke, unutar rječnika `DATABASES` potrebno je definirati podatke o bazi podataka koja će se koristiti - tip baze, ime baze, korisničko ime, lozinku, IP-adresu poslužitelja, port, a moguće je definirati i neke dodatne opcije. Ukoliko se koristi MySQL baza, poželjno je definirati `sql_mode:STRICT_ALL_TABLES` ili `sql_mode:STRICT_TRANS_TABLES` što će osigurati sigurno spremanje novih podataka u bazu te mijenjanje postojećih. Npr. ukoliko se pokuša spremiti novi podatak s vrijednošću polja `age` jednakom 120 (a definirano je da polje `age` smije imati najveću vrijednost jednaku 100) te nije postavljen `sql_mode:STRICT_ALL_TABLES`, novi podatak će biti ubačen s poljem `age` jednakim 100. S druge strane, ako je postavljen `sql_mode:STRICT_ALL_TABLES`, novi podatak neće biti ubačen u tablicu.

```
DATABASES = {
```

```
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'licensing',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': '3306',
        'OPTIONS': {
            'sql_mode': 'STRICT_ALL_TABLES',
        },
    }
}
```

2.2 Stvaranje modela

Svaki model predstavlja jednu tablicu u bazi. Modeli se definiraju u `models.py` datoteci, unutar pojedine aplikacije. Svi modeli su podklasa Djangove ugrađene klase `django.db.models.Model`. Svaki atribut u modelu je zapravo polje (stupac) tablice u bazi.

Primjer modela:

```
from django.db import models

class Company(models.Model):
    company_name = models.CharField(max_length=200, unique=True)
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        verbose_name_plural = "Companies"

    def __str__(self):
        return self.company_name
```

Prethodni model kreirao bi sljedeću tablicu u bazi:

```
CREATE TABLE companies_company(
  id NOT NULL PRIMARY KEY AUTO_INCREMENT,
  company_name VARCHAR(200) UNIQUE NOT NULL,
  is_active TINYINT(1) NOT NULL
  created_at DATETIME NOT NULL
  updated_at DATETIME NOT NULL
)
```

Svakoj tablici automatski je dodijeljeno ime na temelju aplikacije unutar koje se model nalazi te samog modela. U primjeru je ime aplikacije `companies` pa se zato tablica zove

companies_company. Može se zadati i prilagođeno ime tablice tako da se unutar klase Meta modela doda: db_table = moje_ime_tablice. Dodavanje klase Meta nije nužno, a unutar nje se definiraju podaci kao što su ordering, db_name te imena za jedninu i množinu koja su čitljiva čovjeku (verbose_name, verbose_name_plural). Navedeni su samo najčešći parametri koji se definiraju unutar Meta klase. Ukoliko verbose_name i verbose_name_plural nisu definirani, Django će kao ime za jedninu koristiti ime modela preformulirano tako da su sva slova mala te će ispred svakog velikog slova u imenu modela staviti razmak. Ime za množinu će biti jednako imenu za jedninu s dodanim slovom 's' na kraju.

Django svakom modelu automatski dodaje polje:

```
id = models.AutoField(primary_key=True)
```

Ukoliko se želi koristiti drugačiji primarni ključ, dodaje se opcija primary_key=True kod izabranog atributa u modelu. U tom slučaju Django prepoznaje primarni ključ te ne dodaje polje id.

Svaki atribut ima zadanu vrijednost null=False, ako se eksplicitno ne zada null=True (zbog toga su u prethodnom primjeru kreirana polja s NOT NULL ograničenjima). Ako je vrijednost null postavljena na True, Django će prazne vrijednosti u bazu pohraniti kao null vrijednosti. Zadana vrijednost je i blank=False, ukoliko nije definirano drugačije. Vrijednost blank argumenta je vezana uz validaciju. Npr. ako polje ima vrijednost blank argumenta jednaku True, obrazac će prihvatiti i praznu vrijednost tog polja, dok će se u suprotnom vrijednost tog polja morati popuniti.

Atribute u pojedinom modelu jedine je nužno definirati. Osim klase Meta, moguće je definirati i dodatne funkcije koje se koriste na instanci klase koja predstavlja model, kao što je u primjeru definirana funkcija koja vraća stringovnu reprezentaciju modela tj. company_name.

Nakon kreiranja novog modela ili bilo kakve promjene na postojećim modelima, potrebno ih je izvršiti i na bazi. Pokrene se naredba: python manage.py makemigrations companies koja kreira migracije. Rezultat izvođenja navedene naredbe:

```
Migrations for 'companies':
  companies/migrations/0001_initial.py:
    - Create model Company
```

Kreirane migracije trenutno se nalaze unutar migrations direktorija i nisu još izvršene na bazi. Za kreiranje tablice u bazi potrebno ih je izvršiti: python manage.py migrate.

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, companies, sessions
Running migrations:
  Rendering model states... DONE
  Applying companies.0001_initial... OK
```

2.3 Dodavanje novih zapisa u bazu

Putem Python ljuske

```
>>> from companies.models import Company
>>> c = Company(company_name="SML")
>>> c.save()
>>> print(Company.objects.all())
<QuerySet [<Company: SML>]>
```

Putem admin sučelja

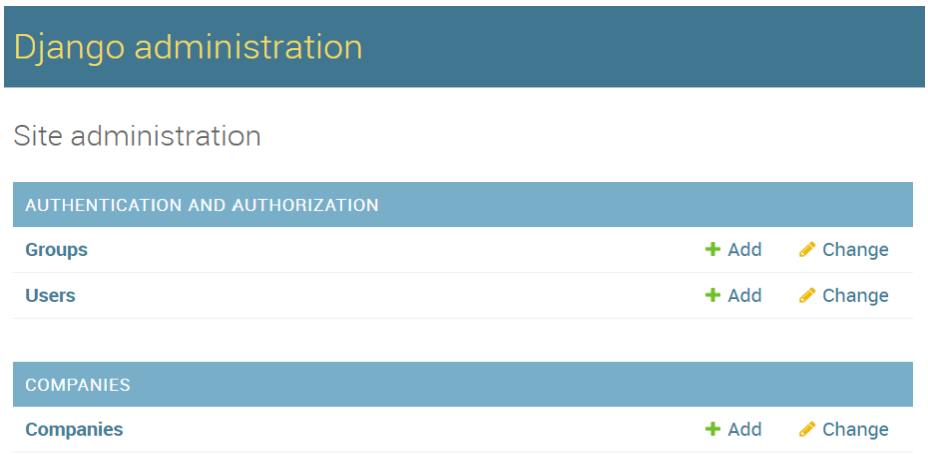
Potrebno je unutar aplikacije u `admin.py` datoteci registrirati modele:

```
from django.contrib import admin
from companies.models import Company

admin.site.register(Company)
```

Nakon toga treba kreirati superkorisnika (eng. *superuser*):
`python manage.py createsuperuser`.

```
Username: matea
Email adress: matea.penezic@gmail.com
Password:
Password (again):
Superuser created successfully.
```

Nakon što se superkorisnik ulogira, na adresi `127.0.0.1:8000/admin/` će vidjeti popis svih postojećih modela koje putem sučelja (2.1) može uređivati, brisati te dodavati nove zapise.



Slika 2.1: Django administracija

Koristeći populacijsku skriptu

Napiše se skripta čijim će se pokretanjem popuniti tablica u bazi. Postoji više biblioteka koje ubrzavaju taj postupak, a jedna od njih je Faker koja ima gotove funkcije za generiranje imena, adresa, datuma, boja i drugih. Prikazana je skripta koja služi za kreiranje instanci modela `Company` te koristi Faker za generiranje imena kompanija.

```
import os
import django
from faker import Faker

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'licensing.settings')
django.setup()

from companies.models import Company

fakegen = Faker()


def populate(n=5):
    for entry in range(n):
        fake_company_name = fakegen.company()

        Company.objects.get_or_create(company_name=fake_company_name,
    uuid=fake_uuid)[0]

if __name__ == '__main__':
    print('populating script...')
    populate(10)
    print('populating complete...')
```

2.4 Polja

Polja su reprezentirana atributima klase koja predstavlja model. Svako polje je instanca `Field` klase. Django koristi tipove polja kako bi odredio: tip stupca u bazi podataka, tj. koji tip podataka će se spremati u toj koloni, zadani HTML widget koji se koristi dok je polje prikazano unutar obrasca (npr. `<select>`) te pravila validacije kod samo-generiranih obrazaca.

Django dolazi s brojnim ugrađenim tipovima polja, a moguće je definirati i prilagođeni tip. U tablici 2.1 prikazano je 10 najčešće korištenih ugrađenih tipova polja.

Tip	Opis	HTML widget
`AutoField`	`IntegerField` koje se samo povećava svaki put prilikom unosa novog podatka u bazu.	
`BooleanField`	Polje koje može imati vrijednost `True` ili `False`; ako je potrebno omogućiti i `null` vrijednost polja koristi se `NullBooleanField`.	`CheckboxInput`
`CharField`	String polje; obavezno je zadati vrijednost argumenta `max_length`. Za velike stringove, odnosno tekst, koristi se `TextField`.	`TextInput`
`DateField`	Obavezni argumenti su `auto_now` i `auto_now_add` čija je zadana vrijednost `False`. Ukoliko je njihova vrijednost `True`, `auto_now` će automatski pohraniti vrijeme u bazu kad je podatak spremljen. `auto_now_add` će automatski pohraniti vrijeme u bazu kad je podatak kreiran.	`TextInput`
`DecimalField`	Decimalni broj; obavezni argumenti su `max_digits` i `decimal_places`.	`NumberInput` ukoliko je `localize=False`, a `TextInput` inače
`EmailField`	`CharField` koje provjerava da je unešena vrijednost email adresa.	`TextInput`
`IntegerField`	Cijeli broj koji može imati vrijednosti od -2147483648 do 214748364.	`NumberInput` ukoliko je `localize=False`, a `TextInput` inače.
`SlugField`	Slug je kratka oznaka koja se sastoji samo od slova, brojeva, znakova '_' i '-'. Obavezan argument je `max_length` čija je zadana vrijednost 50.	`TextInput`
`TextField`	Polje za tekst.	`Textarea`
`URLField`	`CharField` za URL adrese. Ima obavezan parametar `max_length` čija je zadana vrijednost 200.	`TextInput`

Tablica 2.1: Najčešće korišteni ugrađeni tipovi polja u modelu

Svako polje prima skup argumenata specifičnih za taj tip polja. Kao i u prethodnom primjeru s modelom `Company`, postoje i argumenti koji su opcionalni te ih je moguće koristiti na svim tipovima polja. Najčešći su: `null`, `blank`, `choices`, `default`, `primary_key`, `unique`. Slijedi kratko objašnjenje argumenata koji do sada nisu spomenuti:

- `choices` je lista, odnosno tuple parova (2-tuple) koji predstavljaju izbore za polje unutar kojeg se nalaze. Ako je zadan `choices` argument zadani widget unutar obrasca za ovaj atribut će biti padajući izbornik (eng. *select box*). Prvi element u svakom paru je vrijednost koja će biti pohranjena u bazu, dok je drugi element vrijednost koja će biti prikazana u padajućem izborniku.
- `default` označava zadanu vrijednost polja koja se koristi u slučaju spremanja novog zapisa u bazu kada vrijednost tog polja nije eksplictno zadana.
- `unique` znači da ne smije biti više istih vrijednosti tog polja u tablici.

2.5 Relacije

Više-prema-jedan (many-to-one)

Za definiciju više-prema-jedan relacije koristi se `django.db.models.ForeignKey`. Obavezan je pozicijski argument - ime klase s kojom je model u relaciji. Npr. `InstallationFile` je u relaciji više-prema-jedan s `Product` modelom jer jedan produkt može imati više instalacijskih datoteka (budući da ima više verzija), dok instalacijska datoteka može pripadati samo jednom produktu.

```
class Product(models.Model):
    #...

class InstallationFile(models.Model):
    product = models.ForeignKey(Product, related_name='
    installation_files', on_delete=models.CASCADE)
    version = models.CharField(max_length=20, null=True)
    #...
```

Pošto `InstallationFile` model ima `ForeignKey`, u tablici u bazi će se kreirati stupac `product_id`. Ime kolone može se promijeniti definiranjem `db_column` argumenta.

`on_delete` je opcionalan argument, njegova zadana vrijednost je `CASCADE` što znači da će se prilikom brisanja neke instance modela `Product` iz baze automatski obrisati i sve instance `InstallationFile` modela koje su bile u relaciji s obrisanom `Product` instancom.

`related_name` je također opcionalan argument, a predstavlja ime koje se koristi za obrnutu relaciju. U prethodnom primjeru to bi bilo ime za relaciju od `Product` modela prema `InstallationFile` modelu. `related_name` omogućava sljedeće:

```
>>> from products.models import Product, InstallationFile
>>> product = Product.objects.get(id=1)
>>> installation_files = product.installation_files.all()
```

Zadana vrijednost `related_name` argumenta je ime modela kojem je dodan sufiks `_set`.

Vrijednost `related_name` argumenta je ujedno i zadana vrijednost `related_query_name` argumenta koji omogućava obrnutu filtraciju:

```
>>> from products.models import Product, InstallationFile
>>> products = Product.objects.filter(installation_files__version='1.1.1')
```

Više-prema-više(many-to-many)

Za definiciju više-prema-više relacije koristi se `ManyToManyField`. Potrebno je zadati argument koji označava klasu s kojom je model u relaciji. Ako su dva modela međusobno u relaciji više-prema-više, polje `ManyToManyField` smije se nalaziti unutar samo jednog od ta dva modela. Prilikom definiranja `ManyToManyField` Django sam kreira tablicu u bazi koja predstavlja ovu relaciju.

U sljedećem primjeru kreirana je više-prema-više relacija između modela `Order` i `Product`. Model `Order` predstavlja kupovinu (internetsku kupovinu), a model `Product` proizvode. Prilikom jedne kupovine može biti kupljeno više proizvoda te isti proizvod može biti kupljen prilikom više kupovina.

```
class Order(models.Model):
    products = models.ManyToManyField(Product, blank=True)
    #...
```

Primjer dohvaćanja proizvoda neke kupovine i dodavanja novog proizvoda kupovini:

```
order = Orger.objects.get(id=order_id)
products = order.products.all()     #svi proizvodi neke kupovine

product = Product.objects.get(id=product_id)
order.products.add(product)     #dodavanje novog proizvoda kupovini
```

Jedan-prema-jedan(one-to-one)

Za definiciju jedan-prema-jedan relacije koristi se `OneToOneField`. Kao i u prethodnim relacijama, potrebno je zadati pozicijski argument. Ova relacija se najčešće koristi kada model "proširuje" neki drugi model. Zadana vrijednost argumenta `related_name` je ime modela (pisano malim slovima) u kojem se polje nalazi.

Poglavlje 3

Pogledi

3.1 Funkcija pogleda

Funkcija pogleda (eng. *view*), ili samo pogled, je funkcija koja na ulaz dobiva web zahtjev (eng. *request*), nakon toga može tražiti informacije od modela te vraća web odgovor (eng. *response*). Odgovor može biti HTML sadržaj, XML dokument, slika, greška itd. Funkcije pogleda se po konvenciji definiraju unutar `views.py` datoteke.

Slijedi jednostavan primjer pogleda koji na poziv vraća string 'Hello!':

```
from django.http import HttpResponse

def say_hello(request):
  return HttpResponse('Hello!')
```

Svaka funkcija pogleda prima `HttpRequest` objekt kao prvi parametar koji se obično naziva `request`, a vraća `HttpResponse` objekt. Kako bi se ovaj pogled prikazao na određenom URL-u potrebno je definirati taj URL i povezati (mapirati) ga s pogledom u URL konfiguraciji, obično unutar `urls.py` datoteke.

3.2 URL odašiljač

Pogled iz prethodnog primjera može se povezati s URL uzorkom ovako:

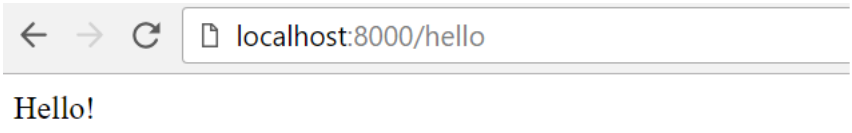
```
from django.conf.urls import url

from . import views

urlpatterns = [
  url(r'^hello/$', views.say_hello),
]
```

Kada dođe novi zahtjev, Django pregledava svaki definirani URL uzorak te staje prilikom prvog podudaranja s URL-om zahtjeva. Nakon toga se poziva pogled koji je povezan s URL uzorkom. Slovo `r` ispred regularnog izraza u URL uzorku označava početak stringa. Nije ga obavezno staviti, ali je poželjno.

Dobra je praksa unutar svake aplikacije imati `urls.py` datoteku u kojoj će se definirati svi URL uzorci za tu aplikaciju te se onda unutar glavnog direktorija u `urls.py` datoteci uvedu svi postojeći URL uzorci iz pojedinih aplikacija. Ako se u prethodnom primjeru radi o aplikaciji `my_app` tada se to može napraviti ovako:

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'', include('my_app.urls')),
]
```

Unosom `http://127.0.0.1:8000/hello` u preglednik pojavit će se poruka 'Hello!' (3.1).



Slika 3.1: Hello

Na sljedećem primjeru objašnjene su grupe regularnih izraza s imenom.

```
url(r'^(?P<product_id>[0-9]+)$', views.installation_files)
```

Sintaksa za grupe regulanih izraza s imenom je `(?P<ime>uzorak)`, gdje je `ime` ime grupe, a `uzorak` predstavlja uzorak s kojim se URL zahtjeva mora podudarati. U ovom primjeru je ime grupe `product_id`, a uzorak `[0-9]+$` (jedna ili više znamenki od 0 do 9 te oznaka za kraj stringa). Sada se osim `request` parametra pogledu `installation_files` prosljeđuje i pozicijski parametar `product_id` pa ga je potrebno drugačije definirati:

```
def installation_files(request, product_id):
```

Svi snimljeni argumenti koji se prosljeđuju pogledu su stringovi, tako da je i `product_id` string.

Funkcija `django.conf.urls.url()` može primiti i treći, opcionalni parametar - rječnik dodatnih argumenata koji se prosljeđuje pogledu. Ako se u prethodni primjer doda kao treći parametar `{'primjer':'prvi'}`, funkcija pogleda onda treba izgledati:

```
def installation_files(request, product_id, primjer):
```

3.3 URL preusmjeravanje

Django pruža mehanizam preusmjeravanja koji omogućava premještanje s jedne URL lokacije na drugu bez upisivanja cijele URL putanje. Za korištenje tog mehanizma nužno je zadati vrijednost `name` argumentu (služi kao identifikacija za pogled koji se poziva) unutar `url()` funkcije prilikom definicije URL uzorka.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    #...
    url(r'^(?P<product_id>[0-9]+)$', views.get_installation_files, name=
    "installation_files"),
]
```

U funkciji pogleda koristi se `reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)` funkcija koja ima obavezan prvi argument - ime pogleda koje je definirano u URL konfiguraciji, ostali argumenti su opcionalni, a u primjeru je naveden `args` argument koji se koristi ako zadani URL prihvaća argumente.

```
from django.http import HttpResponseRedirect
from django.urls import reverse

def edit_installation_file(request, product_id, installation_file_id):
  #...
  return HttpResponseRedirect(reverse('installation_files', args=(
    product_id,)))
```

Unutar predloška navodi se `url` oznaka predloška (eng. *template tag*) ukoliko je korisnika potrebno preusmjeriti na drugi URL.

```
<form action={% url 'installation_files' product.id %} method="post" >
        {% csrf_token %}
        {{ form }}
        <button type="submit" class="btn">
            Submit
        </button
</form>
```

3.4 Povezivanje pogleda i modela

Unutar funkcije pogleda najčešće je potrebno napraviti nešto s podacima u bazi, dohvatiti neke podatke iz nje, promijeniti ih, pohraniti ili ih pak obrisati. To se ne može napraviti direktno, odnosno bez uključivanja modela koji predstavlja tablicu u bazi koja je

potrebna. Nakon što je model uveden u `views.py`, moguće je putem korištenja metoda iz klase `QuerySet` izvršiti potrebne operacije na modelu.

Dostupne metode definirane na klasi `QuerySet` mogu se podijeliti u dvije velike skupine - one koje vraćaju novi `QuerySet` te one koje ne vraćaju `QuerySet`. Metode koje vraćaju `QuerySet` moguće je lančano povezati, tj. unutar istog izraza koristiti više metoda, jednu za drugom. Metode koje ne vraćaju `QuerySet` nije moguće lančano povezati.

U tablici 3.1 dan je kratak pregled najčešće korištenih metoda koje vraćaju `QuerySet`:

Metoda	Opis
`filter()`	Vraća `QuerySet` s objektima koji imaju tražene vrijednosti zadanih parametara.
`exclude()`	Vraća `QuerySet` s objektima koji nemaju tražene vrijednosti zadanih parametara.
`order_by()`	Vraća `QuerySet` s objektima poredanim prema zadanom parametru.
`values()`	Vraća `QuerySet` s rječnicima. Moguće je navesti imena stupaca te će tada biti vraćeni samo njihovi ključevi i vrijednosti.
`all()`	Vraća kopiju trenutnog `QuerySet`.
`select_related()`	Vraća `QuerySet` s objektima koji su vezani putem `ForeignKey`. Metoda se poziva na modelu koji u sebi sadrži polje `ForeignKey`.
`prefetch_related()`	Suprotno od `select_related()` metode. Metoda se poziva na modelu koji drugom modelu predstavlja `ForeignKey`.

Tablica 3.1: Najčešće korištene metode koje vraćaju `QuerySet`

U tablici 3.2 je pregled najčešće korištenih metoda koje ne vraćaju `QuerySet`:

Metoda	Opis
`get()`	Vraća objekt koji ima vrijednost zadanog parametra. U slučaju da postoji više od jednog takvog zapisa u tablici u bazi ili zapis ne postoji, vraća grešku.
`create()`	Služi za kreiranje i spremanje novog podatka.
`get_or_create()`	Ukoliko podatak sa zadanim parametrima postoji u bazi, dohvaća ga, u suprotnom kreira novi podatak s danim parametrima.
`count()`	Vraća broj zapisa u bazi koji odgovaraju danom `QuerySet`-u.
`exists()`	Vraća `True` ako postoji traženi zapis u bazi, u suprotnom vraća `False`.
`update()`	Služi za ažuriranje podataka.
`delete()`	Briše retke iz tablice u bazi koji imaju vrijednosti zadanih parametara.

Tablica 3.2: Najčešće korištene metode koje ne vraćaju `QuerySet`

Kratki primjer povezivanja pogleda s modelom:

```
from .models import Product

def get_products(request):
    products = Product.objects\
                        .filter(company_id=3)\
                        .values('id', 'product_name', 'short_code', '
    valid_duration')\
                        .order_by('product_name')
    #...
```

3.5 Pogledi zadani klasom

Osim funkcijom, pogledi se mogu definirati i klasom. Takva definicija omogućava ponovno upotrebljavanje koda korištenjem nasljeđivanja. Django pruža već gotove generičke poglede (zadane klasom), a moguće je napisati i svoje. U nastavku će se pod riječi pogled podrazumijevati pogled definiran klasom.

Svi pogledi nasljeđuju klasu `django.views.generic.base.View` (ukoliko pogled nije generički klasa, `View` se može uključiti i iz `django.views`). Slijedi primjer kao s početka poglavlja drugačije definiranim pogledom koji ispisuje string 'Hello!'.
`views.py`:

```
from django.http import HttpResponse
from django.views import View

class HelloView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello!')
```

Unutar `urls.py` koristi se `as_view()` metoda za povezivanje pogleda s URL uzorkom:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^hello/$', HelloView.as_view()),
]
```

Ugrađeni pogledi

Ugrađeni pogledi mogu se podijeliti na bazne i generičke. Bazni pogledi predstavljaju roditeljske poglede koji mogu biti korišteni sami za sebe ili ih je moguće naslijediti. Najčešće ih je u projektu potrebno na neki način 'proširiti', odnosno dodati neke dodatne funkcionalnosti. Bazni pogledi su: `View`, `TemplateView` i `RedirectView`. Generički pogledi izgrađeni su na temelju baznih pogleda te služe za automatizaciju nekih često korištenih obrazaca kao što su prikaz liste zapisa iz baze, prikaz detalja o nekom zapisu i sl. Ovoj skupini pripadaju generički pogledi za prikaz (`DetailView`, `ListView`), generički pogledi za editiranje (`FormView`, `CreateView`, `UpdateView`, `DeleteView`) te generički datum pogledi.

Generički pogledi mogu se koristiti direktnim uključivanjem u URL konfiguraciju:

```
from django.conf.urls import url
from django.views.generic import ListView

urlpatterns = [
    url(r'^license/$', ListView.as_view(template_name="license/
    license_list.html")),
]
```

Ili kreiranjem podklase:

```
from django.views.generic import ListView

from licenses.models import License

class LicenseList(ListView):
    model = License
    template_name="license/license_list.html"
```

```
from django.conf.urls import url
from .views import LicenseList

urlpatterns = [
    url(r'^license/$', LicenseList.as_view()),
]
```

Drugi način se koristi kad je potrebno prilagoditi generički pogled vlastitim potrebama što se može napraviti definiranjem odgovarajućih atributa unutar podklase.

Poglavlje 4

Predlošci

4.1 Djangov jezik predložaka

Django pruža jednostavan način generiranja dinamičkih HTML stranica koristeći ugrađene oznake predložaka (eng. *template tags*). Djangov jezik predložaka čine varijable (eng. *variables*) i spomenute oznake.

Varijable se označavaju s: {{ `varijabla` }}. Kada funkcija pogleda proslijedi parametre predlošku, {{ `varijabla` }} se zamjenjuje sa stvarnom vrijednošću te varijable. Ime varijable može biti bilo koja kombinacija slova i brojeva te znaka '_'. Znak '.' koristi se za pristup atributima varijable. Ime varijable ne smije sadržavati razmak kao ni interpunkcijske znakove. Na varijablama se mogu koristiti filtri ({{ `varijabla|filtar` }}) koji služe za modifikaciju prikaza vrijednosti varijable. Između varijable i filtra potrebno je staviti znak `|`. Django pruža oko šezdeset ugrađenih filtra. Npr. filtar `capfirst` će ispisati ime varijable s velikim početnim slovom.

```
{{ varijabla|capfirst }}
```

Na jednoj varijabli može se zadati i više filtra.

```
{{ varijabla|capfirst|default:"Hello" }}
```

`default` predstavlja zadanu vrijednost varijable ukoliko je ona prazan string.

Oznake predloška se označavaju s: {% `oznaka` %}. Slijede neke od najčešće korištenih oznaka:

- `block` ({% `block ime_bloka` %}...{% `endblock` %})
  Označava blok koji može biti predefiniran u predlošku koji nasljeđuje trenutni predložak.
- `extends` ({% `extends "ime_roditeljskog_predloška.html"` %})
  Označava da predložak proširuje roditeljski predložak.

- `for` ({% for objekt in lista %}...{% endfor %})
  Iterira po svim elementima liste omogućavajući pristup elementu u kontekstnoj varijabli. Npr.

```
{% for product in products %}
    {{ product.name }}
{% endfor %}
```

  Moguće je iterirati i po elementima rječnika:

```
{% for key, value in my_dictionary.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

  U prethodnom primjeru `items` nije ključ unutar rječnika `my_dictionary` već metoda za dohvat elemenata.

- `for...empty` ({% for objekt in lista %}...{% empty %}...{% endfor %})
  Unutar for oznake može se nalaziti i `empty` oznaka koja služi za prikaz teksta koji se prikazuje u slučaju kad je lista kroz koju se pokušava iterirati prazna.

```
{% for product in products %}
    {{ product.name }}
{% empty %}
    There are no products.
{% endfor %}
```

  To je ekvivalentno:

```
{% if products %}
  {% for product in products %}
      {{ product.name }}
    {% endfor %}
  {% else %}
    There are no products.
{% endif %}
```

- `if` ({% if produkt %}...{% endif %})
  Ako je izraz koji se evaluira istinit, ulazi se u `if` blok, u suprotnom ga se preskače. Unutar `if` bloka mogu se nalaziti i `elif` i `else` oznake. Može sadržavati i operatore `and`, `or` i `not`, kao i ==, !=, <, >, <=, >=, `in`, `not in`, `is` i `is not` kojih može biti jedan ili više unutar iste `if` oznake.

- `include` ({% include 'ime_datoteke.html' %})
  Služi za uključivanje drugih predložaka u trenutni predložak. Ime predloška može biti ili varijabla ili točno ime datoteke.

- `load` ({% include ime_biblioteke %})
  Uključuje sve oznake i filtre iz dane biblioteke.

- `url` ({% url url_ime parametri... %})
  Vraća apsolutnu putanju (URL bez imena domene) koja se podudara s danim pogledom i opcionalnim parametrima.

Djangov jezik predložaka podržava i komentare koje je potrebno navesti unutar {# #}, npr.

```
{# Ovo je neki komentar... #}
```

4.2 Nasljeđivanje predložaka

Već je spomenuto da se predlošci mogu nasljeđivati. Prvo je potrebno kreirati bazni predložak. Običaj je bazni predložak nazvati `base.html`. Unutar baznog predloška definiraju se blokovi koristeći oznaku `block`. Sadržaj blokova u baznom predloška ne mora biti prazan. Ukoliko je definiran te je definiran sadržaj istog bloka unutar predloška koji ga je naslijedio, bit će prikazan sadržaj tog predloška koji ga je naslijedio. Ako sadržaj istog bloka nije definiran, biti će prikazan sadržaj definiran u baznom predlošku. Svi predlošci koji nasljeđuju bazni predložak moraju imati `extend` oznaku koja se treba nalaziti prije bilo kakve druge oznake. Nakon toga je moguće definirati sadržaj blokova navodeći ime bloka. Opisano se najbolje može vidjeti na primjeru:
`base.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>{% block title %}Django stranica{% endblock %}</title>
</head>

<body>
    <div>
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

`product_list.html`:

```
{% extends "base.html" %}

{% block title %}Products{% endblock %}

{% block content %}
    <table>
```

```
    <thead>
      <tr>
        <th>Product name</th>
        <th>Short code</th>
      </tr>
    </thead>
    <tbody>
    {% for product in products %}
      <tr>
        <td>{{product.product_name}}</td>
        <td>{{product.short_code}}</td>
      </tr>
    {% endfor %}
    </tbody>
  </table>
{% endblock %}
```

4.3 Povezivanje pogleda i predloška

U `settings.py` datoteci unutar `TEMPLATES` liste definiraju se postavke kako bi projekt znao gdje se nalaze datoteke koje predstavljaju predloške. U listi se nalaze rječnici koji sadržavaju postavke za pojedini pogon. Prilikom kreiranja novog projekta putem `startproject` naredbe, Django generira sljedeće postavke:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

U ovom projektu jedino je izmijenjeno:

```
'DIRS': [TEMPLATE_DIR],
```

Pri čemu je:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

Pomoću `BACKEND` definira se koja pozadina predložaka će se koristiti (eng. *template backend*). Navedena pozadina u primjeru je jedna od ugrađenih pozadina. `DIRS` je lista u kojoj se nalaze putanje do direktorija u kojem se nalaze predlošci. `APP_DIRS` označava nalaze li se predlošci unutar instaliranih aplikacija. `OPTIONS` je rječnik koji sadrži neke dodatne parametre koji se prosljeđuju pozadini predložaka. U `context_processors` listi zadaje se koji će kontekst biti proslijeđen svakom predlošku.

Valja istaknuti da se u ovom projektu unutar svake aplikacije nalazi `templates` direktorij unutar kojeg se nalaze poddirektoriji svih postojećih modela definiranih u toj aplikaciji. Dakle, koliko aplikacija ima modela, toliko će `templates` direktorij imati poddirektorija. U poddirektorijima se nalaze predlošci vezani uz odgovarajući model. Također, na razini projekta postoji još jedan `templates` direktorij unutar kojeg je `base.html`. `base.html` je zajednički bazni predložak za sve aplikacije i on predstavlja roditeljski predložak svim ostalim predlošcima u projektu.

Sada kada se želi dohvatiti neki predložak unutar funkcije pogleda ne mora se navoditi cijela putanja do odgovarajućeg predloška:

```
from django.shortcuts import render

def get_products(request):
    #...
    return render(request, 'products/product_list.html', {'products':
    products})
```

Funkcija `render()` će danom predlošku proslijediti navedeni kontekst (eng. *context*) koji je rječnik (u ovom primjeru kontekst je `{'products': products}`) te će vratiti `HttpResponse` objekt (u kojem je predložak popunjen potrebnim elemetnima iz konteksta), na taj način omogućujući prikaz tražene stranice. Ovoj funkciji obavezno je proslijediti parametre `request` i ime predloška, a ostali argumenti su opcionalni.

4.4 Upravljanje statičkim datotekama

Na svakoj web aplikaciji ili stranici se osim čistog HTML-a u pozadini krije i pokoja slika, CSS i Javascript; datoteke koje ih sadrže smatramo statičkim datotekama. Django je za lakše upravljanje takvim datotekama namijenio aplikaciju `django.contrib.staticfiles` koja prikuplja statičke dototeke iz aplikacija. Aplikacija se može ručno dodati unutar `INSTALLED_APPS` liste, unutar `settings.py` datoteke. Ako je projekt kreiran pomoću `startproject` ona bi se trebala već tamo i nalaziti. Nakon što je aplikacija dodana, potrebno je unutar iste datoteke definirati i `STATIC_URL` kako bi Django znao unutar kojeg direktorija se nalaze statičke datoteke. U ovom projektu je to:

```
STATIC_URL = '/static/'
```

Zatim treba kreirati `static` direktorij unutar svake aplikacije. Unutar tog direktorija kreira se direktorij `ime_aplikacije`, unutar kojeg se onda nalaze sve statičke datoteke za tu aplikaciju.

Za uključivanje statičkih datoteka u neki predložak prvo je potrebno pomoću `load` oznake dohvatiti statičke datoteke. Nakon toga kada se želi koristiti neka od dostupnih datoteka potrebno je samo unutar `static` oznake navesti ime željene datoteke.

```
{% load static %}
<img src="{% static "moja_aplikacija/slika.jpg" %}"/>
```

Obično unutar projekta postoje i statičke datoteke koje nisu vezane uz neku aplikaciju. Za njih se se može kreirati direktorij `static` unutar projekta, na istom nivou kao što su i aplikacije, te se unutar `settings.py` definira lista `STATICFILES_DIRS` s putanjom do tog direktorija kako bi Django znao pronaći i te datoteke.

```
STATICFILES_DIR = [os.path.join(BASE_DIR, 'static'), ]
```

U fazi razvoja same aplikacije, potrebno je, osim instalirane aplikacije `django.contrib.staticfiles`, imati postavljeno i:

```
DEBUG = True
```

te će posluživanje statičkih datoteka biti moguće odmah nakon pokretanja poslužitelja pomoću `runserver` naredbe.

Međutim, u fazi produkcije ne koristi se ova metoda, budući da nije efikasna. Većina Django web aplikacija koristi odvojeni web poslužitelj (na kojem nije Django) za posluživanje statičkih datoteka. To funckionira na sljedeći način. Kada dođe do promjene statičkih datoteka, lokalno se pokrene naredba `collectstatic` što će rezultirati skupljanjem statičkih datoteka u direktorij definiran u `STATIC_ROOT`. U `STATIC_ROOT` se nalazi apsolutna putanja do tog željenog direktorija. Potom je potrebno lokalni `STATIC_ROOT` staviti na poslužitelj unutar direktorija koji služi za posluživanje statičkih datoteka.

Poglavlje 5

Django obrasci

5.1 HTML obrasci

HTML obrasci (*eng. HTML forms*) su potrebni za prikupljanje određenih informacije o korisniku koji pregledava stranicu. Nakon što korisnik popuni obrazac, podaci se šalju poslužitelju koji dalje obavlja definirane radnje s obzirom na pristigle podatke.

Obrazac se definira unutar `<form></form>` oznaka. Najvažniji atributi `form` oznake su `method` i `action`. Atribut `method` služi za definicju HTTP metode zahtjeva koji će biti poslan poslužitelju. On može imati vrijednosti `GET` i `POST`. `GET` metoda se koristi kada je potrebno samo dohvatiti neke podatke s poslužitelja, a `POST` metoda se koristi kada će nakon poslanog zahtjeva poslužitelju, eventualno biti napravljene promjene u sutavu, npr. promjene u bazi. Atribut `action` predstavlja skriptu na poslužitelju koja je zadužena za obradu dobivenih podataka iz obrasca.

Elementi obrasca mogu biti polja za unos teksta, padajući izbornici i dr.

5.2 Obrasci u Djangu

Django pruža podršku za obrasce, olakšavajući i ubrzavajući njihovo korištenje. Sve kreće od definiranja obrasca, što se obično radi unutar `forms.py` datoteke koju je potrebno kreirati unutar aplikacije. Slično kao što atributi klase koja definira model opisuju polja u bazi, atributi klase koja definira obrazac opisuju `<input>` elemente HTML obrasca. Obrasci su definirani klasom koja je nasljeđena iz `Form` klase. Slijedi primjer definicije obrasca:

```
from django import forms

class ProductForm(forms.Form):
    product_name = forms.CharField(label='Product name', max_length=100)
```

```
    def clean(self):
        product_name = self.cleaned_data.get('product_name')

        if Product.objects.filter(product_name=product_name).exists():
            raise forms.ValidationError('Product name not available.')
```

Navedeni obrazac će imati jedno polje - `product_name`. Atribut `max_length` označava koliko najviše znakova polje smije imati tako da će preglednik omogućiti korisniku unos najviše toliko znakova. Isto tako će i Django validirati duljinu podatka kada ga primi od preglednika. U gornjem primjeru navedena je definicija `clean()` metode. Ona se najčešće koristi za prilagođenu validaciju. U ovom slučaju se provjerava postoji li takvo ime produkta već u bazi, te se vraća greška u slučaju da postoji.

Cijeli obrazac će izgledati ovako:

```
<label for="product_name">Product name: </label>
<input id="product_name" type="text" name="product_name" maxlength="100"
    required />
```

Oznaka `<form></form>` neće biti generirana tako da ju je potrebno ručno dodati unutar predloška.

Kada preglednik proslijedi Djangu obrazac, odgovarajući pogled je zadužen za njenu obradu. Obično je isti pogled zadužen za objavu praznog obrasca te za obradu popunjenog. Jedan od mogućih pogleda za prethodni obrazac je:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

from .models import Product
from .forms import ProductForm

def get_products(request):
    #...
    form = ProductForm(request.POST or None)
    if form.is_valid():
        product_name = form.cleaned_data['product_name']
        product = Product.objects.create(product_name=product_name)
        product.save()
        return HttpResponseRedirect(reverse('products'))

    return render(request, 'products/product_list.html', {'products':
   products, 'form': form})
```

Ukoliko stigne GET zahtjev, kreirat će se instanca obrasca koja je prazna (`form = ProductForm(None)`) i sprema se u kontekst koji se prosljeđuje predlošku kako bi se prikazao obrazac u pregledniku. Ako stigne POST zahtjev, odnosno pogled primi podatke

ispunjenog obrasca, kreira se instanca obrasca koja je popunjena podacima iz POST zahtjeva (`form = ProductForm(request.POST)`) - podaci se povežu s obrascem.

Slijedi poziv `is_valid()` metode koja služi za validaciju svih polja u obrascu. Ako metoda vrati `False` znači da je poslan ili GET zahtjev ili je poslan POST zahtjev s podacima koji nisu prošli validaciju (metoda `is_valid()` vrati `True` samo ako su sva polja sadržavala ispravne podatke). Ako se radilo o GET zahtjevu korisniku će biti prikazan prazni obrazac. Ako se radilo o POST zahtjevu korisniku će biti prikazan obrazac koji će sadržavati podatke koje je već prethodno unio i potvrdio. Ako metoda `is_valid()` vrati `True` znači da se radi o ispravnom POST zahtjevu. Metoda `is_valid()` u slučaju ispravnih podataka, osim vraćanja `True`, stavi podatke dobivene iz obrasca u `cleaned_data` rječnik.

Nakon što se uđe u `if` , pročitaju se podaci iz rječnika `forms.cleaned_data` te se kreira nova instanca `Product` modela te se spremi u bazu. Na kraju `if`-a se pošalje `HttpResponseRedirect` koji preusmjeri korisnika na drugi URL.

Potrebno je još kreirati odgovarajući predložak:

```
<form action={% url 'products' %} method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

Kao što se vidi, obrazac se u predložak uključuje pomoću {{ `form` }} oznake. Prije obrasca potrebno je staviti {% `csrf_token` %} oznaku kako bi se omogućio POST zahtjev s CSRF zaštitom.

CSRF zaštita

Prilikom kreiranja novog Django projekta moguće je vidjeti da u `settings.py` datoteci postoji ovo:

```
MIDDLEWARE = [
    #...
    'django.middleware.csrf.CsrfViewMiddleware',
    #...
]
```

Pojednostavljeno govoreći `django.middleware.csrf.CsrfViewMiddleware` je klasa koja pruža zaštitu od CSRF napada. CSRF je kartica za *Cross Site Request Forgeries*. CSRF napad je napad kod kojeg neovlaštena osoba pristupa web poslužitelju u ime neke ovlaštene osobe. Npr. pretpostavimo da se korisnik neke web aplikacije, koja služi za objavu članaka, prijavi putem svojeg korisničkog imena i lozinke. Nakon što se prijavio, u novom prozoru otvara preglednik i posjećuje neku stranicu koja je zapravo zlonamjerna i čijim se posjetom automatski šalje zahtjev s popunjenim obrascem koji sadrži

novi članak prema poslužitelju koji servira prethodno spomenutu web aplikaciju. Ako ne postoji CSRF zaštita, poslužitelj će pogledati u sesiju i vidjeti kako je dobio popunjeni obrazac od ovlaštenog, prijavljenog korisnika te će se izvršiti unaprijed definirana skripta (moguće objava novog članka pod imenom napadnutog korisnika). Da je postojala CSRF zaštita, napad ne bi bio moguć. Zaštita funkcionira na sljedeći način. Prilikom prijave korisnika generira se tajna vrijednost i smješta u kolačić (eng. `cookie`). Prilikom svake prijave korisnika generira se nova tajna vrijednost. Na taj način poslužitelj može provjeriti je li zahtjev koji je primio došao sa stranice koja ima odgovarajući kolačić. Ako nije, bit će vraćena greška koja označava da korisnik nema pravo pristupa stranici kojoj pokušava pristupiti. Postoji još jedna tajna vrijednost koja se generira i koja se odnosi na obrasce. Kada korisnik pokuša prvi puta pristupiti nekoj stranici koja sadrži obrazac generira se vrijednost i sprema u `csrfmiddlewaretoken`, skriveno polje obrasca. Svaki put kad na poslužitelj stigne zahtjev s obrascem, Django pogleda `csrfmiddlewaretoken` kako bi provjerio da se radi o odgovarajućem obrascu. U slučaju nepodudaranja tajne vrijednosti ponovo će biti vraćena greška javljajući korisniku kako nema pravo pristupa stranici.

5.3 Kreiranje obrazaca iz modela

Ukoliko su polja prikazana unutar obrasca direktno povezano s definiranim modelima, korisno je koristiti klasu `ModelForm`, koja je podklasa klase `django.forms.Form`. Ona omogućuje jednostavniju definiciju obrasca koristeći zadane atribute u modelu. Prethodno definirani obrazac bi se definirao pomoću `ModelForm` na sljedeći način.
`models.py:`

```
from django.db import models

class Product(models.Model):
    product_name = models.CharField(unique=True, max_length=100, null=
    True, error_messages={'unique': 'Product name not available.'})
```

Kao što se vidi, moguće je navesti rječnik `error_messages` u definiciji modela i u njemu navesti prilagođene poruke koje će se prikazati prilikom pojave neke greške. Tako će se u prethodnom slučaju navedena greška prikazati kada se pokuša spremiti produkt s imenom koje već postoji u bazi. U slučaju da nije eksplicitno navedena poruka za neki tip greške bila bi prikazana ugrađena poruka za taj tip greške.
`forms.py:`

```
from django import forms

from .models import Product

class ProductForm(forms.ModelForm):
    class Meta:
```

```
        model = Product
        fields = ['product_name']
```

Unutar klase `Meta` nužno je definirati model na kojeg se obrazac odnosi. U atributu `fields` navode se atributi modela koji će predstavljati polja u obrascu. Polja u obrascu će biti u onom poretku u kojem su navedena u listi. Ako mu se vrijednost postavi na `'__all__'` svi atributi modela će biti u obrascu. Moguće je postaviti i vrijednost atributa `exclude` ako se želi definirati koji atributi modela ne trebaju biti u obrascu. Polja u obrascu će pokupiti svojstva atributa u modelu. Tako će u ovom primjeru korisnik također moći unijeti maksimalno 100 znakova u polje za ime produkta. Moguće je i predefinirati polja iz modela. Npr. moguće je izabrati drugačiji widget za neko polje navođenjem atributa `widget` ili mu promijeniti oznaku definiranjem `label` atributa. Moguće je i navesti koje će se greške prikazivati (umjesto definiranja grešaka u modelu).

```
from django.utils.translation import ugettext_lazy as _
#...
    class Meta:
        #...
        error_messages = {
            'product_name': {
                    'unique': _('Short product code not available'),
                }
        }
```

Treba jedino imati na umu da veći prioritet imaju poruke greške definirane unutar `Meta` klase nego one definirane u modelu, tako da ako je za istu grešku definirana poruka unutar modela i unutar `Meta` klase, prikazana će biti poruka iz `Meta` klase. U `views.py` bi tada bilo:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

from .models import Product
from .forms import ProductForm

def get_products(request):
    #...
    form = ProductForm(request.POST or None)
    if form.is_valid():
        form.save()
        return HttpResponseRedirect(reverse('products'))

    return render(request, 'products/product_list.html', {'products':
   products, 'form': form})
```

Poglavlje 6

Autentifikacijski sustav

Django ima ugrađen autentifikacijski sustav koji je zadužen i za autentifikaciju i za autorizaciju. Autentifikacija je proces određivanja je li korisnik stvarno osoba za koju se predstavlja. Autorizacija je proces određivanja dopuštenja korisnika. Sustav brine o korisničkim računima, grupama, dopuštenjima i sesijama.

Prilikom kreiranja Django projekta pomoću `django-admin startproject` naredbe, unutar `INSTALLED_APPS` liste, u `settings.py` datoteci, automatski su uključene aplikacije `django.contrib.auth` i `django.contrib.contenttypes`. To su aplikacije koje omogućavaju funkcioniranje Django autentifikacijskog sustava. Također se unutar `MIDDLEWARE` liste nalaze elementi `django.contrib.sessions.middleware.SessionMiddleware` i `django.contrib.auth.middleware.AuthenticationMiddleware` koji su zaduženi za upravljanje sesijama tj. za povezivanje korisnika sa zahtjevom na poslužitelj pomoću sesija.

Navedene postavke će pokretanjem `manage.py migrate` naredbe uzrokovati kreiranje svih potrebnih tablica u bazi koje su vezane uz autentifikacijski sustav.

6.1 User objekt

Centralni dio autentifikacijskog sustava su `User` objekti koji najčešće predstavljaju ljude koji se služe web stranicom. U tablici 6.1 prikazani su atributi `User` objekta:

Polja	**Karakteristike**
`username`	Obavezno polje. Smije imati najviše 150 znakova.
`first_name`	Opcionalno polje. Smije imati najviše 30 znakova.
`last_name`	Opcionalno polje. Smije imati najviše 30 znakova.
`email`	Opcionalno polje.
`password`	Obavezno polje. U bazu se automatski pohranjuje hashirana verzija lozinke.
`groups`	Relacija više-prema-više s `Group` modelom.
`user_permissions`	Relacija više-prema-više s `Permission` modelom.
`is_staff`	Određuje je li korisnik administrator.
`is_active`	Određuje je li korisnički račun aktivan.
`is_superuser`	Određuje je li korisnik superkorisnik.
`last_login`	Datum i vrijeme kada se korisnik zadnji put prijavio.
`date_joined`	Datum i vrijeme kada je korisnički račun kreiran.

Tablica 6.1: Atributi `User` objekta

Novog korisnika može se kreirati pomoću funkcije `create_user()` koja, osim što kreira novi objekt, automatski i sprema kreiranog korisnika u bazu:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('pero777', 'perica@gmail.com', 'neka_lozinka
    ')
```

Novog korisnika također se može kreirati i putem admin sučelja (2.3).

Prilagođeni `User` objekt

Prethodno navedeni `User` objekt ugrađen je u Django, ali ga je moguće i proširiti ili prilagoditi potrebama aplikacije. To se može napraviti na sljedeća četiri načina:

1. **Pomoću Proxy modela**
   Kreira se model koji nasljeđuje `User` model, bez kreiranja nove tablice u bazi. Koristi se kada se želi promijeniti ponašanje postojećeg `User` modela (npr. definiranje novih metoda, zadani poredak...) bez utjecaja na polja modela.

2. **Koristeći jedan-prema-jedan relaciju s User modelom**
   Kreira se model koji će predstavljati novu tablicu u bazi i imat će polje koje predstavlja jedan-prema-jedan relaciju s User modelom. Ovaj način se koristi kada se žele pohraniti neke dodatne informacije o postojećem `User` modelu koje nisu vezane uz autentifikaciju.

3. **Kreiranjem prilagođenog `User` modela proširivanjem `AbstractBaseUser` modela**
   Kreira se potpuno novi user model koji nasljeđuje `AbstractBaseUser` model. Poželjno je kreirati model na početku projekta s obzirom da utječe na shemu baze. U `settings.py` datoteci potrebno je definirati da se ne koristi zadani `User` model, već prilagođeni.

   ```
   AUTH_USER_MODEL = 'moja_aplikacija.MojUserModel'
   ```

   `moja_aplikacija` predstavlja Django aplikaciju unutar koje je model definiran, a `MojUserModel` ime modela koji se koristi kao user model. Ovaj način prilagodbe modela koristi se kada su potrebne izmjene u autentifikacijskom procesu.

4. **Kreiranjem prilagođenog user modela proširivanjem `AbstractUser` modela**
   Kreira se potpuno novi user model koji nasljeđuje `AbstractUser` model. Isto kao i u prethodnom slučaju, poželjno je kreirati model na početku projekta te je potrebno navesti zadani `User` model U `settings.py` datoteci. Ovaj način koristi se kad se ne želi mijenjati ugrađeni autentifikacijski proces već se žele dodati informacije direktno u `User` model, bez kreiranja dodatne klase.

U ovoj aplikaciji koristi se **2.** način prilagodbe `User` modela. Novi model je `Profile` te osim polja koje predstavlja relaciju jedan-na-jedan s User modelom, ima još jedno dodatno polje: `company`. Polje `company` predstavlja relaciju više-prema-jedan s `Company` modelom. Ono je potrebno unutar `User` modela kako bi se znalo kojoj kompaniji pripada pojedini korisnik.

```
from django.db import models
from django.contrib.auth.models import User

from companies.models import Company


class Profile(models.Model):
    user = models.OneToOneField(User, related_name='profile', on_delete=
    models.CASCADE)
    company = models.ForeignKey(Company)

    def __str__(self):
        return self.user.username
```

Model `Profile` imat će odgovarajuću tablicu u bazi. Sada je prilikom kreiranja novog korisnika, potrebno kreirati i novi profil.

6.2 Autentifikacija, prijava i odjava korisnika

Autentifikacija korisnika

Za autentifikaciju korisnika koristi se funkcija `authenticate()` koja kao parametre prima korisničko ime i lozinku te ih provlači kroz autentifikacijske pozadine (*eng. authetication backends*). Prilikom poziva `authenticate()` funkcije, prolazi se kroz pozadine (metode), jednu po jednu, u onom redoslijedu u kojem su definirane. Ako su korisnički podaci valjani za neku od metoda, proces provjere staje te su korisnički podaci valjani. Metode definirane nakon te se ne pozivaju. Ako poziv neke od metoda rezultira `PermissionDenied` iznimkom, ostale metode se ne provjeravaju te autentifikacija ne prolazi. Ukoliko su korisnički podaci valjani, funkcija `authenticate()` vraća `User` objekt, u suprotnom vraća `None` (ako podaci nisu valjani niti za jednu od metoda ili ako je neka od metoda digla `PermissionDenied` iznimku).

Autentifikacijske pozadine definiraju se unutar `AUTHENTICATION_BACKENDS` liste u `settings.py` datoteci. Moguće je definirati i prilagođene pozadine pomoću klase koja sadržava definiciju metoda `get_user()` i `authenticate()`.

U ovom projektu ne koriste se prilagođene pozadine.

Prijava korisnika

Nakon što je korisnik prošao kroz autentifikaciju, potrebno ga je prijaviti u aplikaciju što je moguće napraviti pomoću funkcije `login()`. Funkcija kao parametre prima `HttpRequest` objekt i `User` objekt te sprema korisnika u sesiju.

Slijedi primjer korištenja `authenticate()` i `login()` funkcija:

```
from django.contrib.auth import authenticate, login

def login_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Preusmjeri korisnika na pocetnu stranicu aplikacije.
    else:
        # Prikazi gresku o neuspjesnom logiranju.
```

Odjava korisnika

Za odjavu korisnika iz aplikacije koristi se funkcija `logout()` koja kao parametar prima `HttpRequest` objekt. Funkcija briše korisnika iz sesije.

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
```

U ovoj aplikaciji koriste se ugrađeni autentifikacijski pogledi za prijavu i odjavu korisnika. (vidi 6.3)

Ograničavanje pristupa neprijavljenim korisnicima

Postoji velika vjerojatnost kako će pristup nekim dijelovima web aplikacije biti omogućen jedino prijavljenim korisnicima. Jedan od načina ograničavanja pristupa (koji je korišten i u ovoj aplikaciji) je pomoću `login_required()` dekoratora (funkcije koja prima drugu funkciju kao parametar te joj proširuje funkcionalnost) koji se postavlja prije pogleda za koji se želi ograničiti pristup. Ako je korisnik prijavljen, pogled se normalno izvršava. Ako korisnik nije prijavljen, preusmjerava se na LOGIN_URL adresu definiranu u postavkama. Dekorater se koristi ovako:

```
from django.contrib.auth.decorators import login_required

@login_required
def moj_pogled(request):
    ...
```

Dekoratori se koriste kod pogleda definiranih funkcijom. U slučaju pogleda definiranih klasom koriste se *mixins*. Tako se za ograničavanje pristupa (na samo prijavljene korisnike) pogledu definiranom klasom koristi `LoginRequiredMixin`. Ponašanje je jednako kao kod `login_required()` dekoratora - ako je korisnik prijavljen, pogled se normalno izvršava, a ako korisnik nije prijavljen, preusmjerava se na LOGIN_URL adresu definiranu u postavkama.

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views import View

class moj_pogled(LoginRequiredMixin, View):
    ...
```

Ograničavanje pristupa prijavljenim korisnicima koji ne prolaze test

Ponekad se dijelovima aplikacije želi pružiti pristup samo nekima od prijavljenih korisnika. To se može napraviti pomoću `user_passes_test` dekoratora koji kao argument prima funkciju. Funkciji se prosljeđuje `User` objekt koja vraća `True` ukoliko korisnik smije pristupiti pogledu, a u suprotnom vraća `Flase`.

```
from django.contrib.auth.decorators import user_passes_test

def superuser_or_admin_check(user):
    return user.is_superuser or user.is_staff

@user_passes_test(superuser_or_admin_check)
def my_view(request):
    ...
```

6.3 Autentifikacijski pogledi

Django autentifikacijski sustav pruža nekoliko pogleda koje je moguće koristiti za prijavu korisnika, odjavu korisnika i promjenu lozinke, a to su: `LoginView`, `LogoutView`, `PasswordChangeView`, `PasswordChangeDoneView`, `PasswordResetView`, `PasswordResetDoneView`, `PasswordResetConfirmView`, `PasswordResetCompleteView`. Navedeni pogledi koriste ugrađene obrasce, ali je moguće definirati i prilagođene obrasce. Ugrađeni predlošci za ove poglede ne postoje tako da ih je potrebno kreirati. Moguće je uključiti sve poglede iz ove skupine u projekt:

```
urlpatterns = [
    url('^', include('django.contrib.auth.urls')),
]
```

Time će biti uključeni sljedeći URL uzorci:

```
^login/$ [name='login']
^logout/$ [name='logout']
^password_change/$ [name='password_change']
^password_change/done/$ [name='password_change_done']
^password_reset/$ [name='password_reset']
^password_reset/done/$ [name='password_reset_done']
^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-
    z]{1,20})/$ [name='password_reset_confirm']
^reset/done/$ [name='password_reset_complete']
```

Ili samo neke od njih, kao što se u ovom projektu koriste `LoginView`, `LogoutView` i `PasswordChangeView`:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    url(r'^$', auth_views.LoginView.as_view(template_name='profiles/
    login.html'), name='login'),
    url(r'^logout/$', auth_views.LogoutView.as_view(), name='logout'),
    url(r'^password_change/$', auth_views.PasswordChangeView.as_view(
    template_name='profiles/password_change.html',
```

```
    success_url='/home'),
        name='password_change'),
]
```

U gornjem primjeru prilikom navođenja URL-a za promjenu lozinke (*eng. password change*) naveden je argument `success_url`. On označava URL na koji će se aplikacija preusmjeriti nakon uspješnog procesiranja obrasca. Ukoliko obrazac nije uspješno procesiran, ponovo se prikazuje, ne preusmjeravajući se na drugi URL. URL adrese za preusmjeravanja za prijavu i odjavu korisnika definiraju se unutar `settings.py` datoteke u glavnom direktoriju projekta.

```
LOGIN_REDIRECT_URL = '/home'
LOGOUT_REDIRECT_URL = '/'
```

Poglavlje 7

Web aplikacija za administraciju kupovina putem internetske trgovine softverskih aplikacija

Praktični dio ovog rada je web aplikacija za administraciju kupovina putem internetske trgovine softverskih aplikacija koja demonstrira primjenu principa i pravila Djanga navedenih u prethodnim poglavljima. Ovo je početna i pojednostavljena verzija aplikacije napravljene za tvrtku u kojoj radim.

Web aplikacija je zamišljena kao dio platforme koja bi se se sastojala od nje same te *backend* aplikacije (aplikacija koja se u potpunosti izvršava na poslužitelju) koja bi omogućila integraciju s internetskom trgovinom te softverskim aplikacijama koje se prodaju putem trgovine. Kompanije koje bi koristile platformu, prodavale bi svoje proizvode putem navedene trgovine. Nakon izvršene kupovine *backend* aplikacija bi bila zadužena za spremanje narudžbi, kupljenih proizvoda i pripadnih licenci u bazu. Navedenu bazu bi koristila i web aplikacija za administraciju. Sama aktivacija licence bi se također odvijala putem *backend* aplikacije. U ovom radu napravljena je web aplikacija za administraciju spomenute platforme te se sve na narednim stranicama odnosi na nju.

Web aplikacija ima tri razine prava pristupa ovisno o tipu korisnika (vidi 7.1) i omogućava sljedeće funkcionalnosti:

1. Pretraživanje kupovina softverskih aplikacija, pregled licenci za pripadne produkte (softverske aplikacije) unutar pojedine kupovine, generiranje novih licenci te deaktivaciju licenci.
2. Pregled svih produkata, pripadnih instalacijskih datoteka verzija produkata te njihovo uređivanje.
3. Pregled i uređivanje korisnika.

4. Pregled i uređivanje kompanija.

Projekt se sastoji od tri Django aplikacije: *orders* (7.2), *products* (7.3) i *profiles* (7.4).

Predlošci svih aplikacija nasljeđuju bazni predložak koji se sastoji od navigacijskog izbornika te `div` elementa koji se nalazi na vrhu stranice i sadrži padajući izbornik za izbor kompanije (samo ako je prijavljeni korisnik superkorisnik) te padajući izbornik u kojem se nalaze opcije za promjenu lozinke prijavljenog korisnika i odjavu.

7.1 Tipovi korisnika

Kao što je navedeno, postoje tri tipa korisnika te svaki tip ima različita prava:

- **Obični korisnik** (*user*)
  Obični korisnik može pretraživati kupovine unutar svoje kompanije, pregledati licence unutar pojedine kupovine, genrirati nove licence i deaktivirati licence. Također ima uvid u softverske aplikacije unutar svoje kompanije te u pripadne instalacijske datoteke koje može i uređivati. Može i dodavati nove instalacijske datoteke. Običnom korisniku će u navigacijskom izborniku biti prikazane opcije *Search* i *Products*.

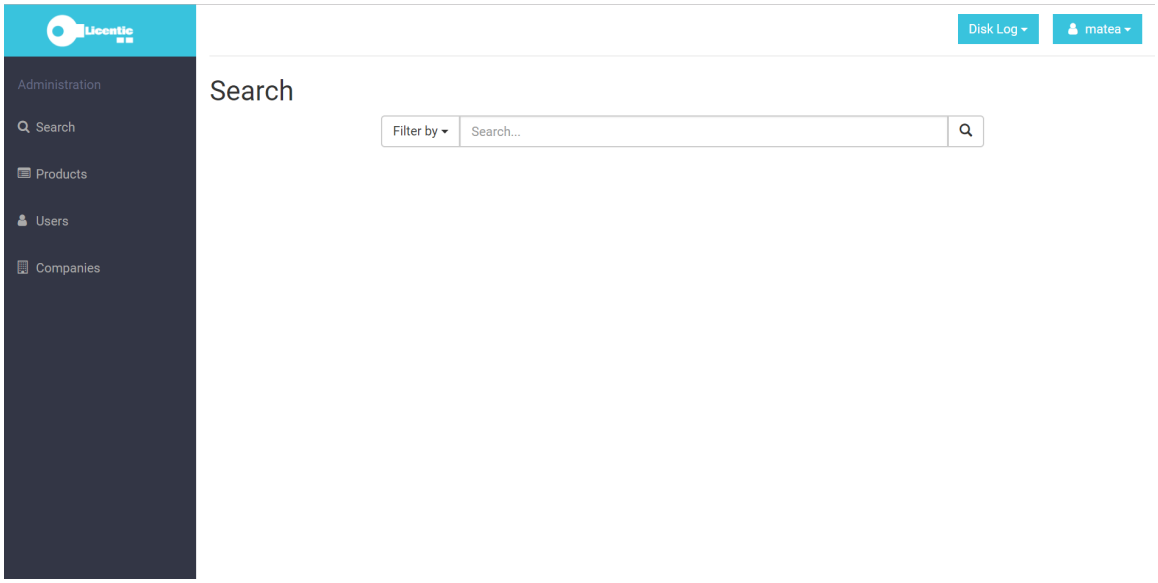
- **Administrator** (*admin*)
  Administrator, osim mogućnosti koje ima obični korisnik, ima pregled u korisnike unutar svoje kompanije. Može uređivati podatke o korisnicima u svojoj kompaniji (osim o superkorisnicima), kao i dodati nove korisnike (osim superkorisnika). Administratoru će u navigacijskom izborniku biti prikazane opcije *Search*, *Products* i *Users*.

- **Superkorisnik** (*superuser*)
  Superkorisnik, osim prava navedenih za prethodne tipove korisnika (s time da može pregledati, urediti i kreirati sve navedeno za bilo koju kompaniju), ima pregled u sve kompanije, mogućnost uređivanja podataka o pojedinoj kompaniji (njenog imena) i dodavanja nove. Superkorisniku će u navigacijskom izborniku biti prikazane opcije *Search*, *Products*, *Users* i *Companies*.

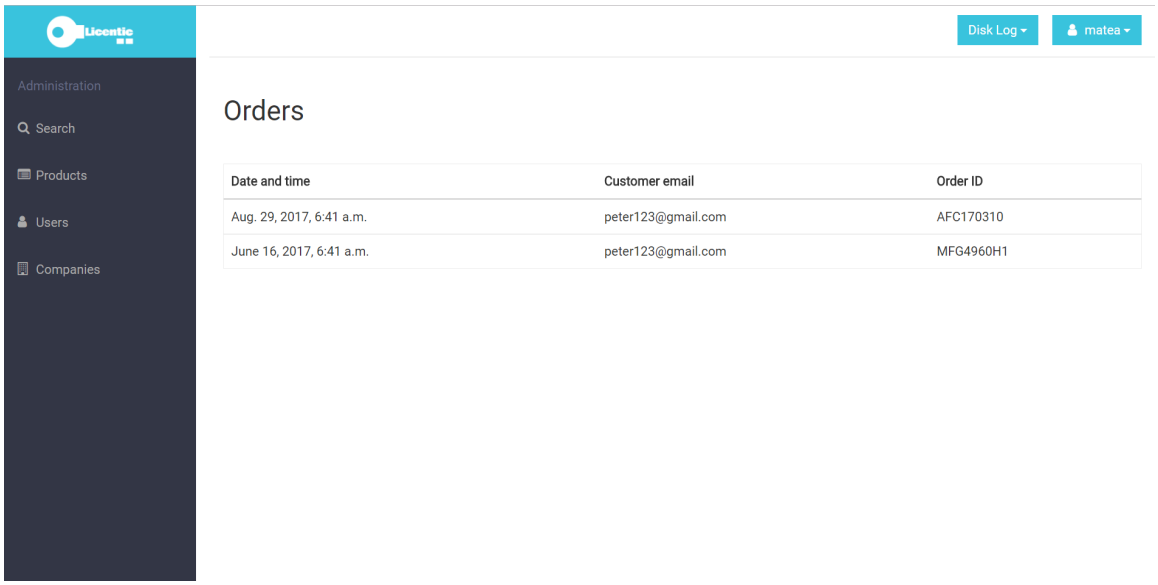
Na svim slikama aplikacije, koje se nalaze u narednim poglavljima, prijavljeni korisnik je superkorisnik.

7.2 *Orders* aplikacija

Aplikacija omogućava pretragu kupovina unosom email adrese kupca (*customer email*) ili identifikacijskog broja kupovine (*order ID*). (7.1, 7.2)

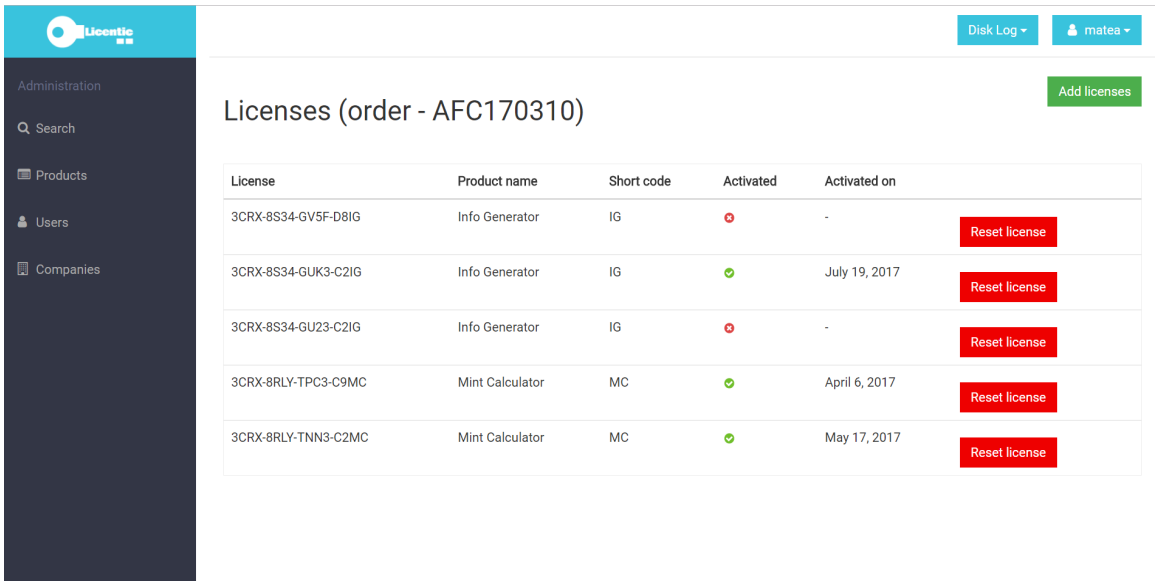


Slika 7.1: Početna stranica web aplikacije, nakon što se korisnik prijavi, je stranica za pretragu kupovina



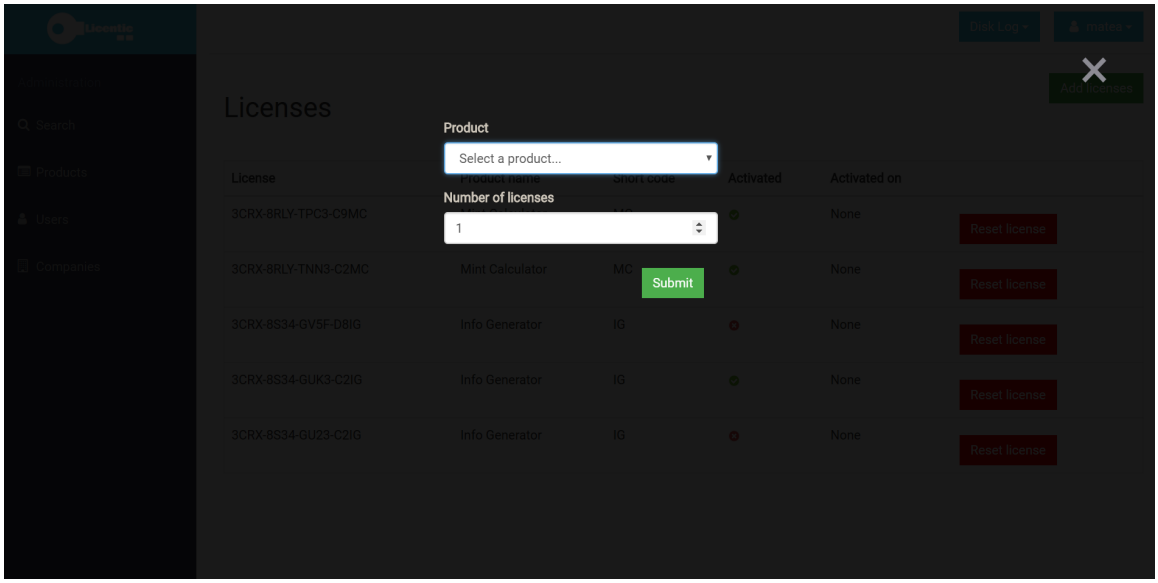
Slika 7.2: Lista svih kupovina za kupca čiji je email peter123@gmail.com

Odabirom pojedine kupovine, prikazuje se lista svih licenci koje su izdane unutar kupovine, zajedno s ostalim podacima o licencama te gumb za deaktivaciju licenci. Licenca je pravo kupca na korištenje kupljenog softvera. Radi pojednostavljenja aplikacije, pretpostavljeno je da se svaka licenca može aktivirati jednom te da traje jednu godinu od aktivacije. Na stranici s listom licenci nalazi se i gumb za dodavanje novih licenci. (7.3)



Slika 7.3: Lista svih licenci za kupovinu AFC170310

Za generiranje novih licenci potrebno je unijeti produkt za koji se želi generirati licence te broj licenci. (7.4).



Slika 7.4: Forma za kreiranje novih licenci

Aplikacija se sastoji od dva modela: `order` i `license`. `Order` model predstavlja kupovinu i sastoji se od atributa:

- **order_id** - identifikacijski broj kupovine (to nije primarni ključ)
- **created_time** - vrijeme kada je izvršena kupovina
- **customer_email** - email adresa kupca
- **company** - `ForeignKey` na model `Company`, označava kojoj kompaniji kupovina pripada
- **products** - `ManyToMany` relacija na model `Product`, kupovina može imati više produkata te se produkt može nalaziti unutar više kupovina.

`License` model ima atribute:

- **order_id** - identifikacijski broj kupovine
- **created_time** - vrijeme kada je izvršena kupovina
- **customer_email** - email adresa kupca

- **company** - `ForeignKey` na model `Company`, označava kojoj kompaniji kupovina pripada
- **products** - `ManyToMany` relacija na model `Product`, kupovina može imati više produkata te se produkt može nalaziti unutar više kupovina.

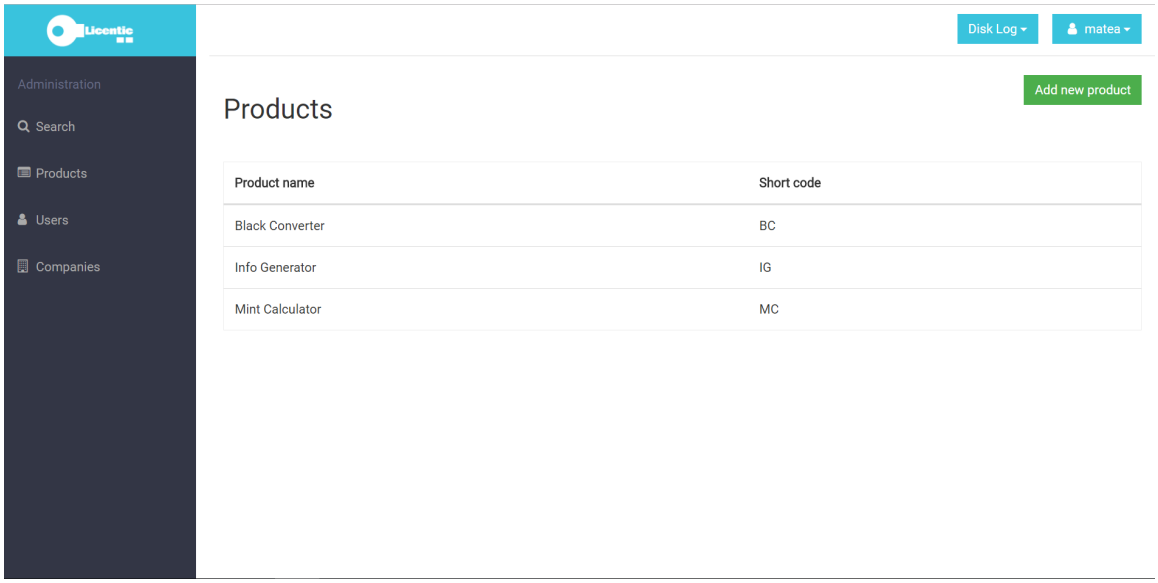
Kada dođe zahtjev za generiranje novih licenci poziva se funkcija `generate_licenses(product_code, quality)` koja poziva funkciju `generate_key(product_code)` onoliko puta koliko se licenci treba generirati. Licence moraju biti jedinstvene. Funkcija `generate_key` koristi pomoćnu funkciju `calculate_integer()` koja na temelju trenutačnog *timestamp-a* u milisekundama, vrijednosti globalne varijable `counter` i slučajno odabranog broja između 0 i 9 kreira veliki pozitivan broj od 16 znamenki. Funkcija `generate_key` konvertira taj broj u bazu zadanu varijablom `base`. Za kreiranje svake licence `generate_licence` će morati pozvati `calculate_integer` dva puta kako bi se postigla željena duljina licence. Na kraju se dobivenom dijelu licence dodaje slučajno odabrani broj između 0 i 9 te `product_code`.

```
from bases import Bases
from random import randint

import time
import math

counter = 0
bases = Bases()


def calculate_integer():
    global counter
    counter += 1
    calculated_id = str(round(time.time() * 1000)) + str(counter) + str(
   randint(0, 9))

    if counter > (math.pow(10, 3) - 1):
        counter = 0

    return int(calculated_id)


def generate_key(product_code):
    key = ''
    count = 0
    randint(0, 9)
    base = '23456789ABCDEFGHJKLMNPQRSTUVWXYZ'
    while len(key) < 16:
        key_part = bases.toAlphabet(calculate_integer(), base)
```

```
        for i in range(0, len(key_part)):

            if len(key) < 16:
                key += key_part[i:i+1]
                count += 1
                if count % 4 == 0:
                    key += '-'
                    count = 0

    key = key + str(randint(0, 9)) + product_code.upper()
    return key


def generate_licenses(product_code, quantity):
    licenses = generate_key(product_code)
    for i in range(1, quantity):
        licenses += '\n' + generate_key(product_code)

    return licenses
```

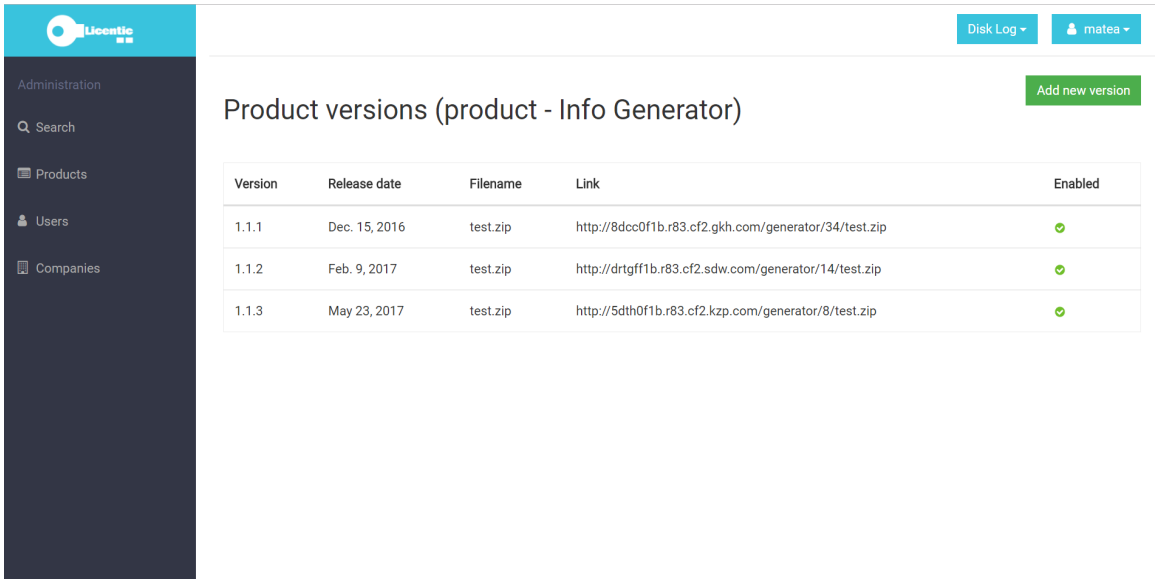
7.3 *Products* aplikacija

Aplikacija služi za pregled produkata te pripadnih instalacijskih datoteka. Odabirom *Products* u navigacijskom izborniku prikaže se lista svih produkata iznad koje se nalazi i gumb za dodavanje novih produkata. (7.5)

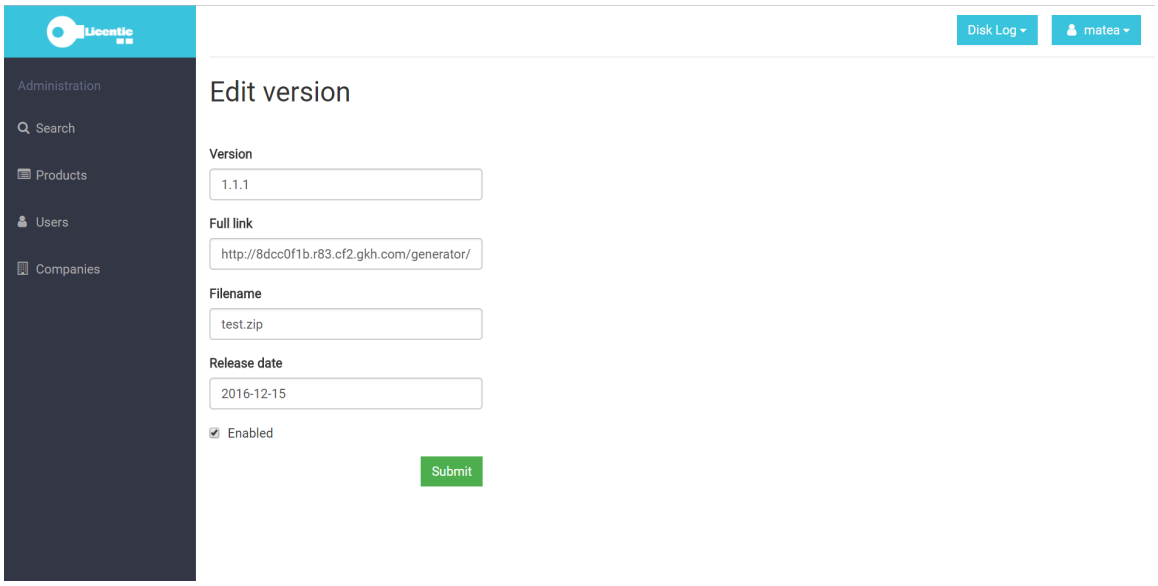


Slika 7.5: Lista svih produkata kompanije *Disk Log*

Odabirom pojedinog produkta prikazuje se lista verzija produkata (iznad koje se nalazi i gumb za dodavanje novih verzija produkata) čiji se podaci mogu uređivati klikom na redak u listi. (7.6, 7.7)



Slika 7.6: Lista svih verzija produkta *Info Generator*



Slika 7.7: Forma za uređivanje verzije

Aplikacija se sastoji od dva modela: `Product` i `Installation file`. `Product` model predstavlja produkt i sastoji se od atributa:

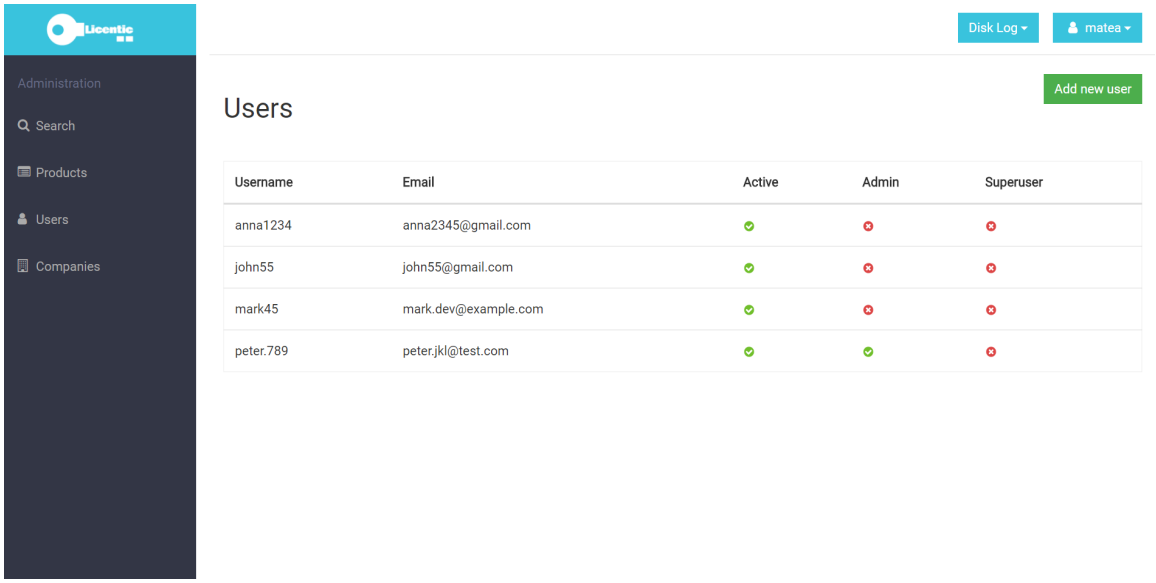
- **product_name** - ime produkta
- **short_code** - kod od dva znaka koji se koristi prilikom generiranja licenci za pojedini produkt; zadnja dva znaka licence predstavljaju *short_code* produkta
- **company** - `ForeignKey` na model `Company`, označava kojoj kompaniji pripada produkt

`InstallationFile` model ima atribute:

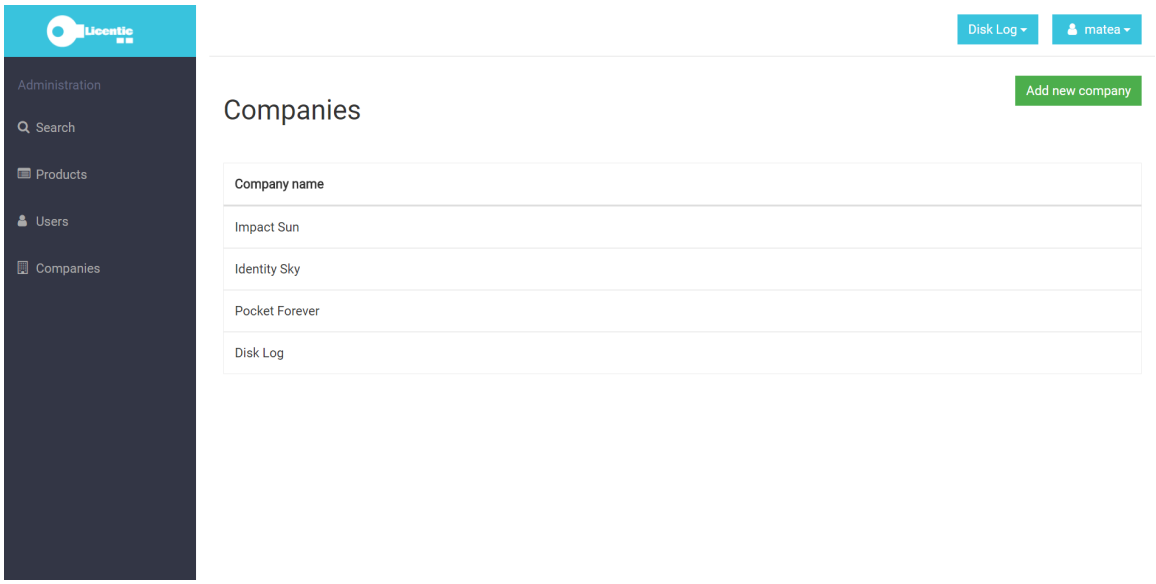
- **product** - `ForeignKey` na model `Product`, kojem produktu pripada instalacijska datoteka
- **version** - verzija produkta
- **full_link** - poveznica putem koje se može skinuti instalacijska datoteka
- **filename** - ime datoteke koja će se skinuti
- **release_date** - datum kada je verzija produkta puštena u produkciju
- **enabled** - je li verzija validna

7.4 *Profiles* aplikacija

Aplikacija omogućava pregled i uređivanje korisnika i kompanija. Na stranicama s pripadnim listama nalaze se gumbi za dodavanje novih korisnika tj. kompanija. (7.8, 7.9)



Slika 7.8: Lista svih korisnika unutar kompanije *Disk Log*



Slika 7.9: Lista svih kompanija

Prilikom kreiranja novog korisnika, šalje mu se poruka na email adresu koja sadrži po-

datke za prijavu u aplikaciju. Moguće je uređivati podatke o pojedinom članu liste klikom na redak.

Aplikacija se sastoji od modela `Profile` i `Company`. `Profile` model je proširenje Djangovog ugrađenog `User` modela te ima atribute:

- **user** - `ForeignKey` na model `User`
- **company** - `ForeignKey` na model `Company`, označava kojoj kompaniji pripada pojedini profil (i korisnik)

Model `Company` sastoji se od samo jednog atributa:

- **company_name** - ime kompanije

Zaključak

Django omogućava brz i pouzdan razvoj web aplikacija te je zbog toga idealan izbor kod projekata koji moraju biti isporučeni u kratkom vremenskom razdoblju. Prati MTV arhitekturu što omogućava preglednu organizaciju i podjelu koda. Dobroj organizaciji pridonosi i podjela Django projekta na aplikacije čineći Django skalabilnim. Također, postoje i brojne vanjske biblioteke (npr. Django REST framework) koje su kompatibilne s Djangom te koje još više ubrzavaju razvoj aplikacije.

Potrebno je neko vrijeme za upoznavanje sa svim Django komponentama te je prije početka razvoja aplikacije potrebno znati kako funkcionira cijeli sustav.

U radu su dotaknute osnovne funkcionalnosti Djanga te je opisano kako povezati sve komponente kako bi aplikacija radila. Također je samo mali dio jednog poglavlja posvećen pogledima zadanim klasom, koji se u praksi koriste sve više i o kojima bi se još dosta moglo reći. Isto tako, u radu nije dotaknuto jedinično testiranje (eng. *unit testing*) za što Django ima podršku koja čini pisanje testova jako jednostavnim. Ima još puno funkcionalnosti koje Django podržava i o kojima bi se moglo pisati.

Svaka nova verzija Djanga donosi neka poboljšanja i unapređenja tako da vjerujem kako će popularnost Djanga u budućnosti sve više rasti.

Bibliografija

[1] *CSRF napadi*, http://www.cert.hr/node/15454.

[2] *Django book*, http://djangobook.com.

[3] *Django documentation*, http://docs.djangoproject.com.

[4] *Django Web Framework*, http://developer.mozilla.org/en-US/docs/Learn/Server-side/Django.

[5] *Django*, https://en.wikipedia.org/wiki/Django_(web_framework).

[6] *HTML Tutorial*, https://www.w3schools.com/html/.

[7] *Learn Django*, http://www.tutorialspoint.com/django/.

[8] *MySQL*, https://www.mysql.com/.

[9] *Python documentation*, http://docs.python.org/3.4/.

[10] *Web framework*, http://en.wikipedia.org/wiki/Web_framework.

[11] Vitor Freitas, *How to Extend Django User Model*, (2016), http://simpleisbetterthancomplex.com/tutorial/2016/07/22/how-to-extend-django-user-model.html.

[12] A. Pinkham, *Django Unleashed*, Sams Publishing, 2015.

Sažetak

U radu je opisan aplikacijski okvir Django koji služi za razvoj web aplikacija. Napisan je u programskom jeziku Python te je besplatan i otvorenog koda. Pojavio se 2003. godine, a od tada mu popularnost sve više raste te se danas ubraja u jedan od najpopularnijih web aplikacijskih okvira.

Slijedi MTV (*model-template-view*) arhitekturalni uzorak. Django projekt sastoji se od jedne ili više Django aplikacije. Svaka od njih predstavlja jednu cjelinu koja obavlja određenu funkcionalnost te se može uključiti i u druge Django projekte.

Logika unutar svake aplikacije podijeljena je na: modele, poglede i predloške te se svaka od tih komponenti zadaje unutar zasebne datoteke. Modeli predstavljaju tablice u bazi. Pogledi su zaduženi za upite prema modelima te se na taj način spremaju ili dohvaćaju podaci. U predlošcima se definira izgled stranice koja će biti prikazana korisniku.

Posljednje poglavlje ovog rada posvećeno je aplikaciji koja je razvijena korištenjem pravila i principa navedenim u prethodnim poglavljima.

Summary

This thesis is about application framework Django which is used for developing web applications. Django is written in Python. It is free and open-source. It was created in 2003. Its popularity has been growing since then and today it is one of the most popular web frameworks.

Django follows MTV (*model-template-view*) architectural pattern. Django project consists of one or more Django applications. Each of them has its own functionality and it is possible to include application in other Django projects.

Logic inside every application can be divided into: models, views and templates. Every one of these components is defined in its own file. Models represent tables in a database. Queries toward models are written in views. Templates define how the page will look like.

Last chapter is dedicated to the appllication which is developed by using rules and priciples mentioned in previous chapters.

Životopis

Rođena sam 11. svibnja 1990. godine u Zagrebu. Nakon završetka osnovne škole upisujem I. gimnaziju u Zagrebu te ju završavam 2009. godine. Iste godine uspisujem Preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku PMF-a u Zagrebu. Završavam ga 2015. godine. U međuvremenu razvijam interes za računarstvo i programiranje te 2015.godine upisujem Diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu.