Slajdovi 1-10

Pogledati video 1 u prilogu predavanja o aplikaciji Talking koja služi kao tekući primer predavanja.

Slajd 11

U nastavku ovog predavanja prelazimo na analizu konkretnih obrazaca za rad sa podacima u veb aplikacijama. Prvi od njih koji ćemo detaljnije proučiti jeste obrazac Query Builder, odnosno graditelj upita.

Slajd 12

Graditelj upita nam pruža veoma pogodan, takozvani "tečan" ili *fluent* interfejs za kreiranje i pokretanje kompleksnih upita nad bazom podataka. U savremenom razvoju softvera, on se koristi za obavljanje većine operacija sa bazom i služi kao odlična alternativa direktnom pisanju upita na jeziku SQL. Sam termin "tečan interfejs" označava objektno-orijentisani idiom koji se zasniva na ulančavanju poziva metoda nad istim objektom. To postižemo tako što svaki metod u lancu kao rezultat vraća referencu na samog sebe, odnosno *this*. Glavni cilj ovog pristupa jeste drastično povećavanje čitljivosti samog programskog koda i praktično kreiranje domenski specifičnog jezika za rad sa podacima.

Slajd 13

Kako bismo ovo bolje razumeli, ilustrovaćemo mogućnosti graditelja upita kroz Django QuerySet API. Prva ključna odlika je ulančavanje metoda, odnosno *chaining*. Upit se ovde gradi korak po korak, pri čemu svaka metoda – poput `.filter()`, `.exclude()` ili `.order_by()` – vraća novi QuerySet objekat. Pogledajmo primer sa slajda: iz klase `Proizvod` pozivamo `.objects.filter` gde izdvajamo elektroniku, zatim ulančavamo `.exclude` da izbacimo artikle kojih nema na stanju, i na kraju dodajemo `.order_by` kako bismo sortirali proizvode po ceni opadajuće. Poenta je jasna: ovo je graditelj jer ne pišemo jedan ogroman SQL string, već programski i elegantno sastavljamo delove upita.

Slajd 14

Druga izuzetno važna karakteristika graditelja upita, posmatrano iz ugla performansi sistema, jeste odloženo izvršavanje, odnosno *Lazy Evaluation*. To znači da se upit

uopšte ne izvršava u samoj bazi podataka sve dok se ti podaci zapravo stvarno ne zatraže u kodu – na primer, kada uđemo u `for` petlju ili eksplicitno pozovemo funkciju `list()`. Praktična korist ovoga je sledeća: vi možete definisati bazični upit na samom vrhu neke funkcije, a zatim kroz različite `if` naredbe dinamički dodavati dodatne filtere, bez ikakvog straha da ćete time više puta bespotrebno "pogoditi" i opteretiti bazu podataka.

Slajd 15

Treća velika prednost je dinamička izgradnja upita. Zamislite scenario gde korisnik na frontend strani popunjava formu za pretragu oglasa i bira različite filtere. U run-time-u, odnosno tokom izvršavanja, mi kreiramo početni upit sa `Oglas.objects.all()`. Zatim, ako je korisnik uneo minimalnu cenu, upit nadograđujemo sa `.filter(cena__gte=cena_od)`. Ako je unet i tekst za pretragu, dodajemo uslov za naslov pomoću `icontains`. Upravo kroz ovakve primere postaje potpuno jasno zašto je Query Builder daleko superiorniji i bezbedniji u odnosu na ručno i mukotrpno spajanje SQL stringova pomoću tekstualnih operacija.

Slajd 16: Razlozi za korišćenje Query Builder-a

Prelazimo na ključne razloge zbog kojih koristimo graditelje upita. Query Builder zapravo nudi idealnu sredinu između previše apstraktnog ORM-a i sirovog SQL-a.

Query Builder nudi četiri glavne prednosti: performanse i kontrolu, sigurnost (automatski binding parametara protiv SQL injection-a), usklađenost sa "Query Objects" trendom, te fleksibilnost za dinamičko građenje upita na osnovu unosa.

Slajd 17: OBRAZAC ACTIVE RECORD (AKTIVNI ZAPIS)

U ovom delu predavanja prelazimo na veoma važan projektni obrazac koji se često koristi u web aplikacijama, a to je obrazac Active Record, odnosno aktivni zapis.

Slajd 18 – Uzorak aktivni zapis

Active Record je arhitektonski obrazac za čuvanje objekata iz memorije u relacionoj bazi podataka. Ideja obrasca je da objekat ne predstavlja samo podatke, već i ponašanje povezano sa radom nad bazom.

Dakle, objekat zna kako da se:

- kreira,

- sačuva,
- ažurira,
- i obriše iz baze.

To su takozvane CRUD operacije – Create, Read, Update i Delete.

U okviru ovog obrasca, svojstva objekta direktno odgovaraju kolonama u tabeli baze podataka. Drugim rečima, jedan objekat predstavlja jedan red u tabeli.

Kada kreiramo novi objekat i pozovemo čuvanje, dodaje se novi red u tabeli. Kada objekat učitamo iz baze, njegova polja se popunjavaju vrednostima iz odgovarajućeg reda. Ako promenimo objekat i ponovo ga sačuvamo, menja se i zapis u tabeli.

Pored osnovnih CRUD operacija, objekti mogu sadržati i sopstvenu domenski specifičnu logiku. Dakle, nisu samo pasivni nosači podataka.

Kod ORM sistema, kao što je Django ORM, uz Active Record dolaze i dodatne mogućnosti:

- relacije među objektima,
- graditelj upita,
- nasleđivanje,
- i razne optimizacije rada sa bazom.

Slajd 19 – Jednostavna realizacija aktivnog zapisa

Ovde vidimo jednostavnu ilustraciju obrasca Active Record.

Imamo baznu klasu Model koja obezbeđuje:

- mapiranje objekta na tabelu baze,
- i osnovne CRUD operacije.

Iz te bazne klase izvodimo konkretne domenske modele.

Na primer, imamo klasu Vest.

Vest poseduje polja:

- naslov,
- tekst,
- i slug identifikator.

Pored podataka, klasa može imati i sopstvene metode poslovne logike.

U primeru je prikazan metod beginning() koji vraća prvih 60 karaktera teksta vesti.

Dakle, model ne sadrži samo podatke, već i ponašanje koje pripada tom domenskom konceptu.

To je jedna od ključnih ideja Active Record pristupa.

Slajd 20 – Django ORM – primer korišćenja aktivnog zapisa

Sada prelazimo na Django ORM kao konkretan primer Active Record obrasca.

Slajd 21

ORM znači Object Relational Mapping, odnosno objektno-relaciono mapiranje.

Problem koji ORM rešava jeste razlika između:

- objektno-orijentisanog sveta aplikacije,
- i relacionog modela baze podataka.

U aplikaciji imamo graf objekata međusobno povezanih referencama.

U bazi imamo skup tabela povezanih stranim ključevima.

ORM predstavlja sloj koji automatski prevodi između ta dva sveta.

Dakle:

- objekti postaju redovi u tabelama,
- relacije među objektima postaju strani ključevi i pomoćne tabele,
- a programer radi sa objektima umesto sa ručno pisanim SQL upitima.

Slajd 22 – Django ORM – kreiranje entiteta

Svaki model u Djangu predstavlja se klasom koja nasleđuje django.db.models.Model.

Svako polje klase predstavlja jedno polje baze podataka.

Ta polja su instance različitih Field klasa. Osnovna odgovornost Field klase je da preslikava python tip nekog polja u tip podataka koji prepoznaje baza (koji sačinjava šemu te tabele u bazi).

Primarni ključ se obično ne navodi eksplicitno jer Django automatski dodaje polje id.

Model može sadržati i domenski specifične metode.

U primeru imamo model klasu User.

Definisana su polja:

- first_name,
- last_name,
- gender,
- username,
- password,
- i druga.

Pored toga postoji i metod assembleDisplayName.

On pravi prikaz imena korisnika na osnovu:

- pola,
- prefiksa,
- imena,
- i prezimena.

Dakle, logika vezana za korisnika nalazi se direktno u modelu.

Slajd 23 – Django ORM – kreiranje entiteta

Ovde vidimo konkretan kod modela User iz Talking aplikacije.

Definisane su konstante:

- FEMALE,
- MALE,
- UNSPECIFIED.

Zatim se definišu moguće vrednosti za pol gender preko choices mehanizma.

Polja modela definišu se pomoću:

- CharField,
- IntegerField,
- i drugih tipova.

Na dnu se nalazi metod assembleDisplayName.

Metod:

- proverava pol korisnika,

- dodaje odgovarajući prefiks,
- dodaje ime i prezime,
- i vraća formatiran prikaz korisnika.

Važno je primetiti da model sadrži i podatke i ponašanje.

Relacije još nisu prikazane. Njih ćemo dodati kasnije.

Slajd 24 – Django ORM – migracije

Kada promenimo modele, potrebno je te promene preneti i u bazu podataka.

To se radi pomoću migracija.

Tipičan tok rada ima tri koraka.

Prvo:

- menjamo models.py.

Zatim:

- pokrećemo komandu makemigrations.

Ona generiše Python skript koji opisuje promene šeme baze.

Posle toga:

- pokrećemo migrate.

Ta komanda izvršava migracije nad bazom bez gubitka podataka.

Migracije omogućavaju evoluciju šeme baze kroz vreme.

Za inicijalne podatke možemo napraviti praznu migraciju pomoću opcije --empty i zatim ručno dopisati funkciju init_data.

Da bi Django znao za aplikaciju, potrebno je:

- dodati aplikaciju u INSTALLED_APPS,
- i definisati AppConfig klasu.

Slajd 25 – Pregled SQL-a migracije

Django može prikazati SQL koji će migracija generisati.

Koristi se komanda sqlmigrate.

Na primeru vidimo SQL za kreiranje tabele djtalk_user.

Django automatski:

- kreira primarni ključ,
- mapira CharField u varchar,
- IntegerField u integer,
- i generiše kompletnu CREATE TABLE naredbu.

To pokazuje kako ORM prevodi objektni model u relacionu šemu baze.

Slajd 26 – Django ORM – CRUD operacije

Sada prelazimo na CRUD operacije.

Za kreiranje objekta:

- napravimo instancu modela,
- popunimo polja,
- i pozovemo save().

U primeru:

- kreira se Post,
- postavlja naslov,
- sadržaj,
- i korisnik kao autor posta.

Korisnik se učitava pomoću:
User.objects.get(id=1)

Nakon toga:
post.save()

Django ORM u pozadini izvršava INSERT nad bazom.

Ovaj kod se tipično nalazi u views.py fajlu.

Slajd 27 – Učitavanje objekata

Objekti se mogu učitavati i iz aplikacije i iz Django shell-a.

Komanda:
Post.objects.all()

vraća QuerySet svih postova.

Nad QuerySet-om možemo iterirati kao nad kolekcijom.

Za svaki post možemo pristupati njegovim poljima kao običnim atributima objekta.

To je jedna od glavnih prednosti ORM pristupa – programer radi sa objektima umesto sa rezultatima SQL upita.

Slajd 28 – Filtriranje objekata

Django ORM omogućava veoma moćno filtriranje.

Za nalaženje jednog objekta koristimo:
get()

Za filtriranje:
filter()

Za isključivanje:
exclude()

Filteri se mogu ulančavati.

Uslovi imaju oblik:
field__lookup=value

Na primer:
user__first_name='Petar'

znači na primer dohvati sve postove:

- koje je napisao korisnik Petar
- i proveri uslov da naslov posta ne sadrži reč computing.

Na isti način možemo koristiti:
contains,
icontains,
gte,
lte,
i druge operatore.

Slajd 29 – Izmena i brisanje objekata

Za izmenu objekta dohvatimo objekat iz baze:

- promenimo polja,
- i pozovemo save().

Za brisanje:

- nad objektom koji je dohvaćen iz baze pozivamo delete().

Dakle, objekat sam zna kako da izvrši promenu nad bazom.

Slajd 30 – Django ORM – relacije

Talking aplikacija koristi više vrsta relacija:

- 1:1,
- 1:N,
- M:N,
- jednosmerne,
- dvosmerne,
- i samoreferišuće relacije.

Django ORM pruža veoma jednostavan način za definisanje svih tih relacija.

Relacije su u djangu navigabilne sa obe strane.

Takođe, preko relacija možemo pristupati objektima u više koraka.

Na primer:

- od korisnika do prijatelja,
- od prijatelja do njegovih postova,
- pa do tagova tih postova.

Django podržava i različite oblike nasleđivanja među modelima.

Slajd 31 – Relacija 1:1

Relacija jedan-prema-jedan definiše se u model klasi pomoću OneToOneField.

U primeru:
UserInfo pripada tačno jednom User objektu.

Polje:
user = models.OneToOneField(...)

automatski generiše strani ključ user_id koji je istovremeno i primarni ključ tabele UserInfo.

Time se obezbeđuje da jedan korisnik ima najviše jedan UserInfo zapis.

Slajd 32 – Relacija 1:N

Relacija jedan-prema-više realizuje se pomoću ForeignKey.

U primeru:
jedan User može imati više Post objekata.

Zato model Post sadrži:
user = models.ForeignKey(User,...)

Django automatski generiše strani ključ user_id u tabeli post.

Slajd 33 – Relacija M:N

Relacija više-prema-više realizuje se pomoću ManyToManyField.

U primeru:
jedan Post može imati više Tag objekata,
a jedan Tag može pripadati više postova.

Django automatski generiše pomoćnu tabelu post_tag.

Ta tabela sadrži:

- post_id,
- i tag_id.

Programer ne mora ručno da pravi tu tabelu, kao što bi bio slučaj da bazi pristupa putem SQL-a direktno.

Slajd 34 – Inverzna strana relacije

Django automatski generiše pristup i sa druge strane relacije.

Ako Post ima ForeignKey ka User,
onda iz User objekta možemo pristupiti svim postovima preko:
user.post_set.all()

Ako želimo drugačije ime,
u ForeignKey(...) koristimo related_name parametar.

Talking aplikacija sadrži i samoreferišuće relacije:

- lifePartner,
- i myFriends.

Njih je preporučljivo dodatno proučiti u kodu aplikacije.

Slajd 35 – Nasleđivanje u Django ORM-u

Django podržava više načina nasleđivanja.

Prvi pristup je multi table inheritance.

U tom slučaju:

- bazna klasa ima svoju tabelu sa svojim poljima,
- svaka podklasa ima dodatnu tabelu samo sa svojim dodatim poljima,
- i povezane su stranim ključevima.

Drugi pristup je apstraktna bazna klasa.

Tada:

- bazna klasa nema tabelu,
- samo njene podklase imaju tabele,
- a nasleđena polja se kopiraju u podklase.

Treći pristup su proxy klase.

One ne menjaju šemu baze (samo bazna klasa se pamti), već samo dodaju ponašanje.

Slajd 36 – Primer nasleđivanja

U Talking aplikaciji:

- Post je bazna klasa,
- a VideoPost i ImagePost su izvedene klase putem multi-table nasleđivanja.

VideoPost dodaje:
videoURL

ImagePost dodaje:
imageURL

Django automatski pravi posebne tabele:

- djtalk_videopost,
- i djtalk_imagepost.

Primarni ključ podklase ujedno je i strani ključ ka tabeli Post.

Slajd 37 – Korišćenje nasleđivanja

Ovde vidimo rad sa VideoPost objektom.

VideoPost:

- ima sva polja Post klase,
- plus videoURL.

Objekat VideoPost automatski ima identitet Post objekta.

Može se koristiti i kao Post.

Preko post_ptr možemo pristupiti baznom objektu.

Preko post.videopost možemo uraditi downcast ako je objekat zaista video post, inače se baca izuzetak.

Slajd 38 – Korišćenje relacija

Relacije se koriste kao dinamička polja.

Na primer:
iz korisnika možemo dobiti njegove postove.

Django podrazumevano koristi lazy loading.

To znači da se povezani objekti učitavaju tek kada prvi put pristupimo relaciji.

To može dovesti do problema N+1 upita, jer za svakog korisnika imamo poseban upit da dovučemo njegove postove.

Zato koristimo:
prefetch_related()

Tako možemo unapred učitati povezane objekte i smanjiti broj SQL upita, jer tako nalažemo djangu da kada dovlači iz baze postove, odmah dovuče i povezane tagove za svaki post u drugom primeru.

Slajd 39 – Ažuriranje relacija 1:1

Kod relacija 1:1 važno je pozvati save() nad samim povezanim objektom.

Dakle:
user.contactdata.save()

a ne:
user.save()

Jer izmene postoje u ContactData objektu, a ne u User objektu.

Slajd 40 – Ažuriranje relacija M:N

Za M:N relacije koristi se add().

Objekti moraju već postojati u bazi.

Primer:
entry.authors.add(a1, a2, a3)

Django automatski ažurira pomoćnu tabelu relacije.

Slajd 41 – Uklanjanje relacija

Za uklanjanje svih elemenata iz relacije koristi se:
clear()

Za uklanjanje pojedinačnih objekata:
remove()

Važno:
time se briše samo relacija,
a ne sami objekti iz baze.

Slajd 42 – Transakciona obrada

Transakcija predstavlja skup operacija koje se izvršavaju atomično.

Dakle:

- ili se sve izvrše,
- ili se nijedna ne izvrši.

Django koristi AUTOCOMMIT režim.

To znači da je svaki save() posebna transakcija.

Opcija ATOMIC_REQUESTS omogućava da cela obrada HTTP zahteva bude jedna transakcija.

To povećava konzistentnost,
ali može uticati na performanse.

Slajd 43 – Eksplicitne transakcije

Django omogućava ručno upravljanje transakcijama pomoću:
transaction.atomic

Može se koristiti:

- kao dekorator,
- ili kao context manager.

Ako nema izuzetka:
radi se commit.

Ako dođe do greške:
radi se rollback.

Slajd 44 – Problem sa finderima

Ovde vidimo problem konzistentnosti objekata u memoriji.

Moguće je imati dva različita memorijska objekta u aplikaciji koji predstavljaju isti red baze (imaju isti ključ id).

To može dovesti do konflikata prilikom čuvanja, čak i za istog korisnika.

Rešenje za taj problem je obrazac Identity Map.

Drugo moguće rešenje je pažljivo korišćenje transakcija i Unit of Work obrasca, čime se rešavaju problemi višekorisničkog pristupa u našoj veb aplikaciji.

Slajdovi 45,46 – Obrazac Identity Map

Identity Map obezbeđuje da se svaki objekat učita samo jednom tokom obrade zahteva.

Objekti se prilikom prvog dovlačenja iz baze čuvaju u mapi prema identitetu (ključ je id i vrsta objekta).

Kada se ponovo traži isti entitet, vraća se već postojeća instanca.

Slajd 47 – Identity Map u Djangu

Biblioteka django-idmap omogućava dodavanje Identity Map obrasca.

Modeli tada nasleđuju:
IdMapModel

Time se obezbeđuje:

- deljenje istih instanci,
- i izbegavanje duplog učitavanja objekata.

Slajdovi 48,49 – Obrazac Unit of Work

Unit of Work prati:

- nove objekte,
- izmenjene objekte,
- i obrisane objekte.

On koordinira upis svih promena u bazu.

Cilj je:

- smanjenje broja pristupa bazi,
- rešavanje konkurentnosti,
- i održavanje konzistentnosti.

Slajd 50 – Kako funkcioniše Unit of Work

Jedinica rada zna:

- koji su objekti učitani,
- koji su promenjeni u memoriji u odnosu na bazu (dirty),
- i koje treba sačuvati.

Kada dođe commit:

- otvara transakciju,
- proverava konkurentnost,
- i upisuje promenjene (dirty) objekte.

Programer ne mora ručno da vodi računa o redosledu SQL operacija.

Slajd 51 – Pozivalac registruje promenu

U ovom pristupu,
pozivalac eksplicitno registruje objekat kao promenjen (dirty) kod Unit of Work komponente.

Pri commit-u, Unit of Work prodje kroz sve promenjene objekte i sačuva ih u bazi.

Slajd 52 – Objekat sam sebe registruje

Alternativno,
objekat može sam sebe registrovati.

Na primer:

- setter metode označavaju objekat kao izmenjen,
- a učitavanje ga registruje kao čist objekat.

Slajdovi 53, 54 – Obrazac Data Transfer Object

DTO je obrazac za prenos podataka između procesa ili slojeva sistema.

Cilj je smanjenje broja udaljenih poziva, jer to usporava sistem.

Umesto mnogo malih poziva,
šalje se jedan objekat koji sadrži sve potrebne podatke, dakle ne samo objekat nego rezultat celog upita za koji se pretpostavlja da će se često koristiti (u primeru i objekat Album i povezani objekat Artist).

DTO mora biti:

- jednostavan,
- serijalizabilan (da podatke čuva u pogodnom formatu za prenos, na primer JSON),
- i pogodan za prenos preko mreže.

Za potrebe pakovanja objekta u serijalizabilan format na predajnoj strani i raspakivanja na prijemnoj, koristi se instana asembler klase koja ima upravo to zaduženje.

Slajd 55 – Karakteristike DTO-a

DTO obično:

- agregira podatke iz više objekata,
- koristi jednostavna polja,
- i izbegava složene grafove objekata (koji postoje u samoj aplikaciji kada koristimo ORM).

Najčešće se koristi za JSON komunikaciju između backend-a i frontend-a.

Slajd 56 – DRF Serializer kao DTO

U Django REST Framework-u šablon DTO implementira osnovna klasa Serializer koja ima sledeće uloge:

- DTO,
- validator,
- i assembler.

Serializer:

- validira podatke,
- pretvara JSON u objekat,
- i objekat u JSON.

Primer primene serijalizatora nad običnim python objektom (Poruka nije instanca model klase).

Metod create() pravi domenski objekat iz validiranih podataka (na prijemnoj strani). Sama klasa serializer ima ugradjen metod za serijalizaciju (uokvireni primer).

Slajd 57 – DRF ModelSerializer

ModelSerializer koji je u DRFu izveden iz osnovnog Serializera automatizuje rad sa modelima (instancama model klasa).

On:

- automatski generiše polja,
- povezuje serializer sa modelom,
- i dodaje validacije poput unique ograničenja.

Slajd 58 – Primer iz Talking aplikacije

Ovde vidimo kompletan primer REST API-ja.

Model:
Post

Serializer:
PostSerializer

View:
PostViewSet

Router automatski generiše CRUD rute:

- list,
- create,
- retrieve,
- update,
- destroy.

Dakle, veoma brzo možemo napraviti kompletan REST API, uz pisanje samo ovih nekoliko linija koda.

Slajd 59 – Zaključak: Put podatka kroz obrasce

Podatak u modernom web sistemu prolazi kroz više slojeva.

U bazi:

- podatak je relacioni zapis.

U ORM sloju:

- postaje objekat preko Active Record obrasca.

U memoriji:

- Identity Map i Unit of Work održavaju konzistentnost.

Kod formiranja upita:

- Query Builder omogućava fleksibilno kreiranje upita.

Na granici sistema:

- DTO i serializeri pretvaraju podatke u JSON i nazad.

Slajd 60 – Finalni zaključak

Ključna poruka ovog predavanja jeste razdvajanje odgovornosti.

Promene u jednom sloju sistema ne bi trebalo da ruše druge slojeve.

Na primer:

- promena baze ne treba da zahteva promenu frontenda,
- a promena prikaza ne treba da menja model baze.

Što su granice između slojeva jasnije definisane,
to je arhitektura stabilnija i lakša za održavanje.