

Projektovanje veb aplikacija - uzorci za podatke -

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

Uvod

- Konceptualni model aplikacije u objektno orijentisanom projektovanju realizuje se u vidu takozvanog **domenskog modela** koji obuhvata i podatke (koji se čuvaju u bazi) i ponašanje (funkcije).

Domenski model

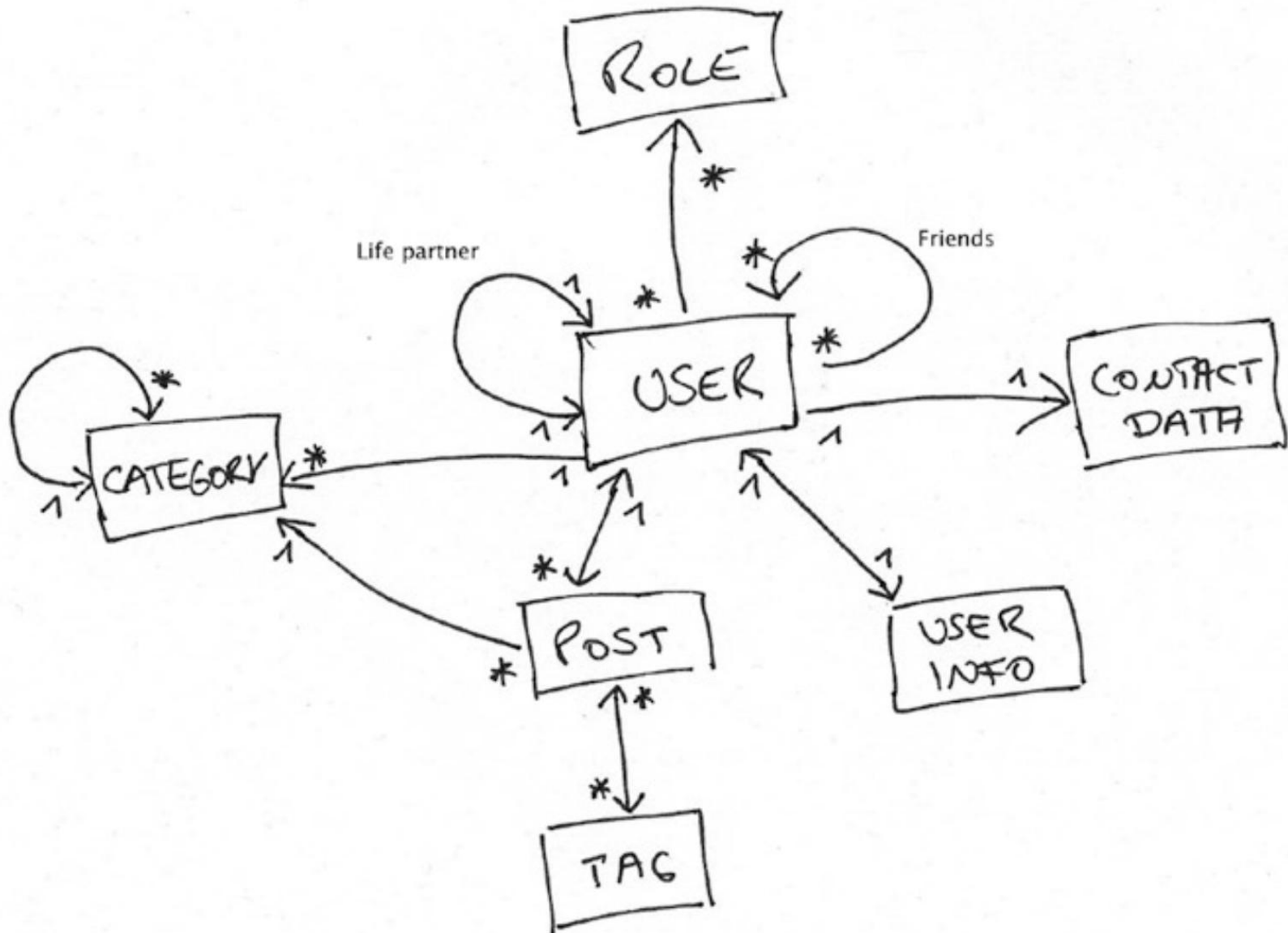
Čine ga:

- **Entiteti** -- Klase i objekte koji predstavljaju glavne koncepte domena.
 - U internet prodavnici, glavni elementi bi biti „Kupac“, „Narudžba“, „Proizvod“, „Korpa“ i tako dalje. Domenski specifične klase i objekti se koriste umesto generičkih kolekcija podataka, kao što su PHP nizovi kad god je to moguće.
- **Vrednosni objekti** u poređenju sa entitetima, nemaju postojani identitet, u smislu primarnog ključa u bazi podataka (njihovo glavno svojstvo je vrednost, na tome se zasniva poređenje jednakosti dva objekta).
- **Međusobne veze (asocijacije)** između domenskih klasa i objekata.
 - U primeru online prodavnice, jedna porudžbina bi imala najmanje jednog Kupca, kao i jednu ili više referenci na naručene proizvode.
- **Poslovna logika** implementiranu u okviru domenskih objekata aplikacije kad god je to moguće, a ne na primer, u kontrolerima (kao deo MCV šablona).
 - U internet prodavnici Korpa, na primer, može imati metod izracunajUkupnuCenu(). na osnovu artikala u Korpi i njihove količine.
- **Servisi** -- Funkcije koje obuhvataju više entiteta
 - U online prodavnici servis „Checkout“ vodi računa o smanjenju zaliha, fakturisanju, ažuriranju istorije narudžbina itd. Usluga se bavi sa više entiteta odjednom.

Prednosti korišćenja domenskog modela

- Odvajanje dela aplikativnog koda koji je specifičan za konkretnu primenu (dakle domenski specifičnog koda) od ostatka koda koji rešava generalne tehničke probleme (kao što je prikaz GUIja, logging, zaštita podataka, slanje pošte itd)
 - olakšava održavanje koda i njegovo menjanje,
 - povećava prenosivost koda (na primer, mogućnost njegovog prebacivanja na novi okvir).
- Promoviše timski rad i između programera i korisnika, menadžera i marketinga.
 - Domenski model prevazilazi razlike u razmišljanju inženjera i ne-inženjera tako što unifikuje terminologiju.

Tekući primer ovog predavanja



Tekući primer – domenski model

- U demo aplikaciji Talking, korisnik (**User**) može pisati postove. **Post** uvek ima samo jednog autora (Korisnika). Korisnik može biti u jednoj ili više uloga (**Role**), ali nije predviđeno da se vrši pretraga korisnika koji imaju određenu ulogu.
- Korisnik može da pristupi zapisu **UserInfo** koji sadrži datum registracije i datum odjavljivanja, ako su dostupni. Na osnovu UserInfo podataka mogu se pretraživati korisnici.
- Korisnik pristupa zapisu **ContactData** koji čuva Korisnikov e-mail i telefonski broj. Na osnovu ContactData ne pretražuju se korisnici.
- Korisnik može navesti drugog korisnika kao svog životnog partnera (relacija **LifePartner**). Korisnikov životni partner je dvosmerna relacija.
- Korisnik može imati neograničen broj prijatelja (**Friends**). Za zadatog korisnika, mogu se pretražiti njegovi prijatelji, ali nije predviđena pretraga u suprotnom smeru.
- Postovi korisnika mogu imati neograničen broj oznaka (**Tags**). Oznaka se može ponovo koristiti u nekoliko postova. Između Posta i njegovih Oznaka postoji dvosmerno povezivanje.

Tekući primer – domenski model

- Post referiše svoju **Kategoriju**, međutim, iz Kategorije nije predviđeno referisanje njenih postova. Kategorija može imati potkategorije, kao i nadređenu Kategoriju, ako je zadata. Kategorije su korisnički specifične. Korisnik navodi svoje kategorije, ali nema povratnog referisanja, od Kategorije ka Korisniku.
- Ovo je relativno mali model, ali reprezentativan jer poseduje različite vrste relacija među entitetima: 1:1, 1:N, M:N, unidirekzione, bidirekzione i samoreferišuće relacije (sa obe strane relacije je isti entitet).
- Dodatno ćemo u model ubaciti relaciju (jednostrukog) nasleđivanja, kroz razradu da post može biti čisto tekstualni ali može imati i dodatak u vidu ili audio ili video zapisa (ne i jednog i drugog). Dakle Post je nadklasa, a ona se proširuje u dve podklase: **AudioPost** i **VideoPost**, gde se na polja osnovne klase, dodaje polje audioUrl, odnosno videoUrl.

Tekući primer – slučajevi korišćenja

- Pošto je primarni cilj ove aplikacije ilustracija projektovanja sloja podataka, definisani su samo neki osnovni scenariji rada aplikacije:
 - Na osnovnoj strani prikazuje se spisak sažetaka postova sa naslovom, prva dva reda sadržaja, autorom i dugmetom za čitanje punog posta
 - Strana punog posta prikazuje izabrani post naslovom, punim sadržajem, autorom, spiskom tagova ako su definisani (hiperlink koji vodi na stranu sa spiskom postova za dati tag), kategorijom i akcijama nad postom Edit i Delete.
 - Strana za ažuriranje posta prikazuje formu u kojoj se mogu izmeniti naslov i sadržaj postjećeg posta i ima dugme Add Post za čuvanje izmena.
 - U zaglavlju aplikacije postoje linkovi za osnovnu stranu (Talking), za podatke o tekućem korisniku (My posts&Profile) i dodavanje novog posta.
 - Na strani My Posts&Profile prikazuje se ime korisnika, lista naslova njegovih postova, lista prijatelja i ime životnog partnera i hijerarhijska lista kategorija koje je korisnik definisao (ako je definisao). Naslovi postova i životni partner su hiperlinkovi, vode na odgovarajuće strane.
 - Na strani za dodavanje novog posta prikazuje se forma u kojoj se mogu definisati naslov i sadržaj i ima dugme Add Post za čuvanje unetog.

Tekući primer - realizacija

- Primer je realizovan u tri varijante:
 - Django aplikacija po **ModelViewTemplate** uzorku. Ova realizacija je primer korišćenja projektnog obrasca **Aktivni zapis** (eng. **Active Record**) na osnovu koga radi Django ORM (objektno-relacioni mapper)
 - Django aplikacija gde se koristi Django REST (primer obrasca **Data Transfer Object**) i django templejti za prezentacioni deo.
 - Django aplikacija sa Django REST, klijentski deo odvojeno kao React aplikacija
- Ove aplikacije nalaze se na sajtu predmeta kao prilog predavanju.

OBRAZAC QUERY BUILDER (GRADITELJ UPITA)

Projektni obrazac Graditelj upita

- **Graditelj upita** pruža pogodan, “tečan” (engl. **fluent**) interfejs za kreiranje i pokretanje upita u bazi podataka.
- Može se koristiti za obavljanje većine operacija sa bazom podataka u aplikaciji kao alternativa pisanja upita na jeziku SQL.
- Tečan interfejs je objektno-orijentisani idiom koji se zasniva na ulančavanju poziva metoda nad istim objektom (što je omogućeno time što svaki metod vraća `this`).
- Cilj je povećavanje čitljivosti programa i kreiranje domenski specifičnog jezika.

Ilustracija QueryBuilder-a

Mogućnosti Django QuerySet API-ja:

1. "Chaining" (Ulančavanje metoda)

Upit se gradi korak po korak. Svaka metoda (poput `.filter()`, `.exclude()`, `.order_by()`) vraća novi QuerySet objekat, što omogućava slaganje logike:

Ilustracija Query Builder-a kroz Django QuerySet

```
upit = Proizvod.objects.filter(kategorija='Elektronika') \
    .exclude(na_stanju=0) \
    .order_by('-cena')
```

Poenta: Ovo je "Builder" jer ne pišemo jedan ogroman SQL string, već programski sastavljamo delove upita.

Ilustracija QueryBuilder-a

Mogućnosti Django QuerySet API-ja:

2. Lazy Evaluation (odloženo izvršavanje)

Ovo je ključna karakteristika Query Builder-a u kontekstu performansi. Upit se **ne izvršava** u bazi dok se podaci zapravo ne zatraže (npr. u for petlji ili pozivom list()).

- **Primer:** Možeš definisati bazični upit na vrhu funkcije, a zatim dodavati filtere kroz if naredbe bez brige da ćeš "pogoditi" bazu više puta.

Ilustracija QueryBuilder-a

Mogućnosti Django QuerySet API-ja:

3. Dinamička izgradnja upita

Scenario gde se filteri grade u run time-u na osnovu korisničkog unosa sa frontenda:

```
upit = Oglas.objects.all()
if cena_od:
    upit = upit.filter(cena__gte=cena_od)
if pretraga:
    upit = upit.filter(naslov__icontains=pretraga)
```

Ovdje studentima postaje jasno zašto je Query Builder superioran u odnosu na ručno spajanje SQL stringova.

Razlozi za korišćenje Query Builder-a

Nudi sredinu između previše apstraktnog ORM-a i sirovog SQL-a:

- 1. Performanse i kontrola:** Koristi se kada standardni ORM postane prespor ili previše komplikovan za specifičan upit.
 - 2. Sigurnost:** Automatski rešava problem **SQL injection** napada kroz binding parametara, što je ključno u modernom razvoju.
 - 3. Trend "Query Objects":** Umesto teških "Repository" slojeva korišćenje lakših QueryObjects koji direktno koriste Query Builder-e za specifične potrebe.
 - QueryObject praktično pakuje neki kompleksniji QuerySet upit u posebni klasu – ima metod za upit i čuva i neke od parametara upita (zato se instancira kao objekat).
- **Fleksibilnost:** Neophodan je u projektima gde se filteri i uslovi upita grade dinamički na osnovu korisničkog unosa sa frontenda.

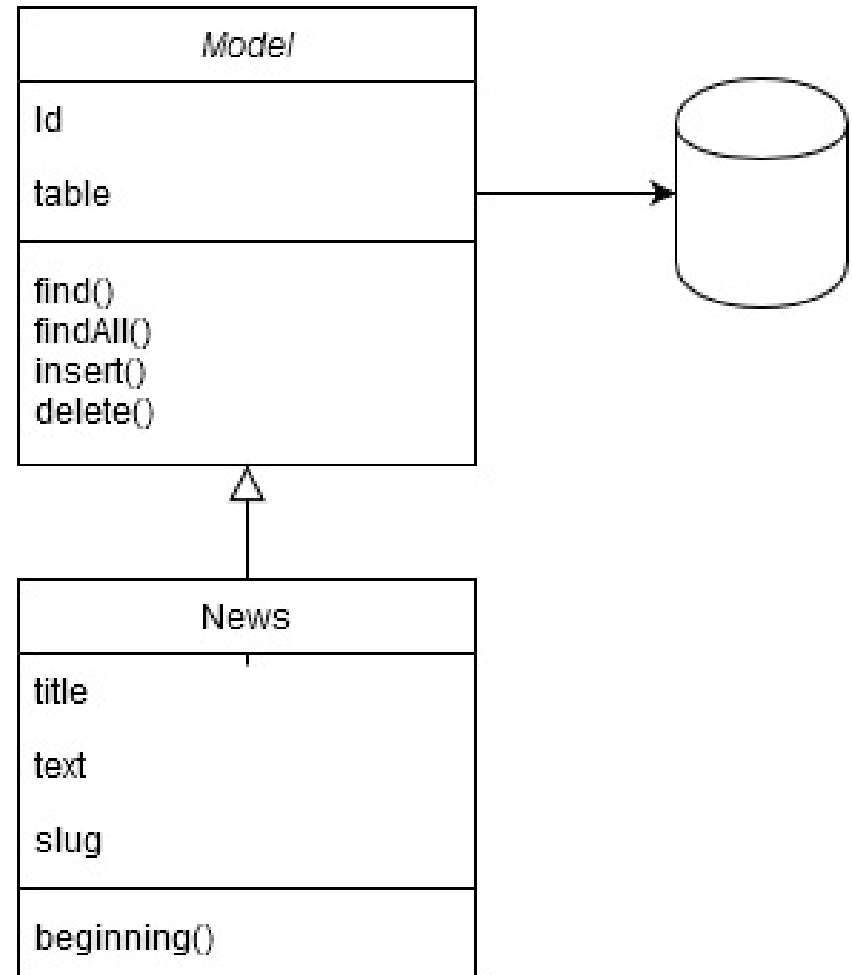
**OBRAZAC ACTIVE RECORD
(AKTIVNI ZAPIS)**

Uzorak aktivni zapis

- **Aktivni zapis** (eng. **Active Record**) je arhitektonski projektni obrazac za čuvanje podataka o objektu u memoriji u relacionoj bazi podataka. Interfejs objekta u skladu s ovim obrascem uključuje funkcije kao što su kreiranje, umetanje, ažuriranje i brisanje objekta u bazi (CRUD), plus svojstva koja manje ili više direktno odgovaraju kolonama u tabeli baze podataka.
- Prema ovom obrascu instanca objekta vezana za jedan red u tabeli. Nakon kreiranja objekta, novi red se dodaje u tabelu. Svaki učitani objekt dobija svoje podatke iz baze podataka. Kada se objekt ažurira, odgovarajući red u tabeli se takođe ažurira. Klasa Model koja “umotava” pristup bazi implementira pristupne metode ili svojstva za svaku kolonu u tabeli ili prikazu.
- Pored ovog osnovnog ponašanja, objekat može dodatno da implementira sopstveno ponašanje (domenski specifično), a u nekim sistemima objektno-relacionog mapiranja, mi ćemo proučiti django orm kao primer, dodaju se relacije među objektima, graditelj upita itd.

Jednostavna realizacija aktivnog zapisa

- Model je klasa koja obezbeđuje osnovno ponašanje: mapiranje stanja objekta na red u tabeli baze podataka i CRUD operacije.
- Iz nje izvodimo konkretne modele za naše domenske objekte koji mogu dodati specifično ponašanje.
- U našem primeru, Vest ima osobine naslov, tekst, slug-identifikator za Wordpress sajt. Takođe ima i specifični metod `beginning()` koji vraća prvih 60 znakova teksta vesti.



DJANGO ORM

- primer korišćenja aktivnog zapisa -

- ORM je skraćenica od **Object Relational Mapping** tj. objektno-relaciono mapiranje.
- Ovi sistemi rešavaju problem kako zapamtiti (perzistirati) domenski model koji se realizuje u vidu grafova međusobno povezanih objekata, u relacionoj bazi koja se realizuje u vidu kolekcije međusobno povezanih relacionih tabela.

Django ORM – kreiranje entiteta

- Svaki model (entitet koji će se čuvati u bazi) predstavljen je klasom koja je podklasa `django.db.models.Model`. Svaki model ima određeni broj polja klase, od kojih svaka predstavlja polje baze podataka u modelu.
- Svako polje je predstavljeno instancom klase `Field`(ne navodi se primarni ključ koji je po konvenciji `id`).
- Model može sadržati i proizvoljne metode domenski specifične logike koji obrađuju podatke modela. U primeru klase `User` metod `assembleDisplayName` pravi string reprezentaciju imena korisnika sa prefiksom (`gdin` ili `dja`) koji zavisi od pola.

Django ORM – kreiranje entiteta

```
# djtalk/models.py
```

```
from django.db import models
```

```
class User(models.Model):
```

```
    FEMALE = 1
```

```
    MALE = 2
```

```
    UNSPECIFIED = 0
```

```
    GENDER_CHOICES = [(UNSPECIFIED, 'Unspecified'), (FEMALE, 'Female'), (MALE, 'Male'), ]
```

```
    first_name = models.CharField('First name', max_length=255)
```

```
    last_name = models.CharField('Last name', max_length=255)
```

```
    gender = models.IntegerField(default=UNSPECIFIED, choices = GENDER_CHOICES)
```

```
    name_prefix = models.CharField('Salutation', max_length=255)
```

```
    username = models.CharField('User name', max_length=255)
```

```
    password = models.CharField(max_length=255)
```

```
    . . . .
```

```
    def assembleDisplayName(self):
```

```
        displayName = ""
```

```
        if self.gender == self.MALE:
```

```
            displayName += 'Mr.'
```

```
        if self.gender == self.FEMALE:
```

```
            displayName += 'Mrs.'
```

```
        if self.name_prefix != "":
```

```
            displayName = self.name_prefix
```

```
            displayName += ' ' + self.first_name + ' ' + self.last_name
```

```
        return displayName
```

- Primer klase User iz aplikacije Talking. Nisu prikazane relacije, to ćemo naknadno specificovati u modelu.

Django ORM – kreiranje entiteta

- Da bi se promene u modelu prenele u bazu (kreirale ili promenile šemu tabela u bazi), potrebno je kreirati migraciju (opis promena šeme) a zatim primeniti migraciju nad bazom. Dakle generalno se ponavljaju sledeća tri koraka:
- Editovanje modela u **models.py**
- Komanda **python manage.py makemigrations** pravi odgovarajući python skript, inicijalno je to 0001_initial.py
- Komanda **python manage.py migrate** izvršava najsvježiji migracioni skript i pravi promene šeme baze bez brisanja podataka
- Za inicijalne podatke u bazi može se prvo napraviti prazan migracioni skript komandom: **python manage.py makemigrations --empty yourappname** a zatim u njemu napisati funkcija `def init_data(apps, schema_editor)`
- Pogledati u primeru projekta migracioni skript 0002
- Da bi komande radile, potrebno je da u settings.py fajla django sajta postoji linija:

```
INSTALLED_APPS = [  
    'djtalk.apps.DjtalkConfig',
```
- a da u djtalks/apps.py postoji klasa `DjtalkConfig` izvedena iz `AppConfig` sa poljem `name = "djtalk"`

Django ORM – kreiranje entiteta

- Da bismo pregledali sql kod nekog migracionog skripta, npr. 0001_initial.py bez izmena baze, zadajemo komandu
- **python manage.py sqlmigrate djtalk 0001**

```
BEGIN;
```

```
--
```

```
-- Create model User
```

```
--
```

```
CREATE TABLE "djtalk_user" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
"first_name" varchar(255) NOT NULL, "last_name" varchar(255) NOT NULL,  
"gender" integer NOT NULL, "name_prefix" varchar(255) NOT NULL,  
"username" varchar(255) NOT NULL, "password" varchar(255) NOT NULL);  
COMMIT;
```

Django ORM – CRUD operacije

Kreiranje novog objekta prema model klasi.

```
from .models import Post, User
```

```
post = Post()  
post.title = 'title'  
post.content = 'text'  
post.user = User.objects.get(id=1)  
post.save()
```

- Ovaj deo koda tipično se nalazi u modulu views.py. Django pogledi odgovaraju kontrolerima MVC projektnog uzorka.
- Imena modela koji koristimo moramo importovati iz models.py
- Pravi se novi objekat entiteta, postavljaju se polja na odgovarajuće vrednosti i pozivom save() metoda vrši njegov upis u povezanu tabelu baze.
- Frejmwork u pozadini pristupa bazi (podrazumevano sqlite). Pristupni parametric se podešavaju u **djsite\settings.py**

Django ORM – CRUD operacije

Učitavanje iz baze postojećeg objekta

- Upute je sem u aplikaciji, moguće zadavati i u Django shell-u
- Učitavanje svih objekata tipa Post (dobija se kolekcija kroz koju je moguće normalno iterirati:

```
C:\djsite>python manage.py shell
```

```
(InteractiveConsole)
```

```
>>> from djtalk.models import Post
```

```
>>> posts = Post.objects.all()
```

```
>>> for post in posts:
```

```
...     print(post.title)
```

```
...     print(post.content)
```

Django ORM – CRUD operacije

Učitavanje iz baze postojećeg objekta

- Nalaženje jednog objekta prema primarnom ključu:

```
>>> from djtalk.models import Post
>>> post = Post.objects.get(id=1)
>>> print (post.title)
```

- Nalaženje upitom sa uslovom (za detalje pogledat Django dokumentaciju Making queries). Sa all() se dohvataju svi objekti, sa filter(uslov) oni koji zadovoljavaju, a sa exclude(uslov) oni koji ne zadovoljavaju uslov, sa get(uslov) jedan određeni objekat. Filteri mogu da se ulančavaju da se specifikuje složeni uslov.
- Osnovni uslov je **field__lookuptype=value**

```
>>> from djtalk.models import Post
>>> post = Post.objects.filter(user__first_name = 'Petar').exclude(title__contains = 'computing')
>>> post
<QuerySet [<Post: Post object (2)>]>
>>> post[0].title
'Jurassic Park'
```

Django ORM – CRUD operacije

Izmena postojećeg entitetskog objekta

- dovoljno je promeniti vrednosti polja i zatim pozvati `save()` kada treba upisati izmene u bazu. Napomena: ako treba sačuvati i objekat i njegovu M:N relaciju onda se koristi `add()` umesto `save()`.

```
>>> from djtalk.models import Post
>>> post = Post.objects.get(title="Proba")
>>> post.title = "Novi naslov"
>>> post.save()
```

Brisanje entitetskog objekta

- Kada je entitet prethodno pronađen (učitan), uklanjanje se vrši pozivom metoda `delete()` :

```
>>> post.delete()
```

Django ORM – relacije

- U domenskom modelu Talking aplikacije definisali smo niz relacija među entitetima. Relacije se karakterišu **kardinalnošću** i **usmerenjem**. Prema kardinalnosti, relacije mogu biti 1:1 (na primer, entiteti user i user_info), 1:N (na primer, entiteti user i post), M:N (entiteti Post i Tag). Prema usmerenju, relacije mogu biti jednosmerne (navigabilne sa jedne strane, npr. od User-a može pristupiti kategorijama koje je definisao, ali obrnuto ne očekujemo) i dvosmerne (navigabilne sa obe strane), na primer, možemo dobiti sve Postove nekog Usera, ali i za određeni Post možemo dobiti njegovog autora.
- Objektima entiteta možemo pristupati i preko više relacija, na primer, za korisnika možemo naći njegovog prijatelja, a zatim naći post tog prijatelja i njihove tagove.
- Django pruža jednostavan način za definisanje različitih relacija 1:1, 1:N, M:N i drugih i u principu sve te relacije su navigabilne sa obe strane.
- Nasleđivanje je takođe jedna od relacija među entitetima. Django ima direktnu podršku za nekoliko vrsta nasleđivanja.

Django ORM – definisanje relacija

1:1 relacija (User ↔ UserInfo)

- u model UserInfo uvodi se OneToOneField, navodi se klasa u relaciji User i da će polje koje realizuje relaciju a koje će po konvenciji biti nazvano user_id ujedno biti primarni ključ tabele UserInfo.

```
class UserInfo(models.Model):
```

```
    user = models.OneToOneField(User, on_delete=models.CASCADE,  
                                primary_key=True)
```

```
    signUpDate = models.DateTimeField('Date published')
```

```
    signOffDate = models.DateTimeField('Date published')
```

```
--
```

```
-- Create model UserInfo
```

```
--
```

```
CREATE TABLE "djtalk_userinfo" ("user_id" bigint NOT NULL PRIMARY KEY  
REFERENCES "djtalk_user" ("id") DEFERRABLE INITIALLY DEFERRED,  
"signUpDate" datetime NOT NULL, "signOffDate" datetime NOT NULL);
```

Django ORM – definisanje relacija

1:N relacija (User ⇔ Post)

- Za definisanje ove relacije uvodimo polje tipa ForeignKey u model Post. Parametar je ime modela User u relaciji.

```
class Post(models.Model):
```

```
    user = models.ForeignKey(User, on_delete=models.CASCADE)
```

- Izaziva generisanje stranog ključa u tabelu post radi podrške relaciji

```
CREATE TABLE "djtalk_post" (...  
"user_id" bigint NOT NULL REFERENCES "djtalk_user" ("id")  
DEFERRABLE INITIALLY DEFERRED);
```

Django ORM – definisanje relacija

M:N relacija (Tag ⇔ Post)

- Za definisanje ove vrste relacija koristi se `ManyToManyField` u jednom od modela sa navođenjem modela sa kojim se pravi relacija. Nije važno u koji od modela se stavlja ovo polje.

```
class Post(models.Model):  
    tag = models.ManyToManyField(Tag)
```

- Automatski se generiše tabela `post_tag` radi podrške relaciji

```
CREATE TABLE "djtalk_post_tag" (  
"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
"post_id" bigint NOT NULL REFERENCES "djtalk_post" ("id")  
DEFERRABLE INITIALLY DEFERRED,  
"tag_id" bigint NOT NULL REFERENCES "djtalk_tag" ("id")  
DEFERRABLE INITIALLY DEFERRED);
```

Django ORM – definisanje relacija

Inverzna strana relacije

- Ako je definisana neka relacija, na primer

class Post(models.Model):

```
user = models.ForeignKey(User, on_delete=models.CASCADE)
```

- onda se može pristupiti iz objekta klase User svim postovima notacijom **user.post_set.all()**, dakle ime povezanog modela uz dodato `_set`.
- Ako se želi posebno imenovati reverzna strana, u originalnom ForeignKey ili ManyToMany polju dodati **related_name='zeljeno_ime'**

Ostale vrste relacija

- Primer Talking aplikacije poseduje još nekoliko zanimljivih vrsta relacija. Postoje samo-referišuće relacije `myFriends` i `lifePartner` nad entitetom `User`. Preporučuje se njihovo samostalno proučavanje u kodu aplikacije.

Django ORM – nasleđivanje

- Postoje tri načina da se (jednostruko) nasleđivanje entiteta realizuje u Doctrine-u:
 1. **Multi table inheritance** – u ovoj varijanti u tabeli natklase se pamti sadržaj natklase i sopstvena polja natklase. Svaka potklasa ima dodatnu tabelu čiji je primarni ključ ujedno i strani ključ iz tabele natklase i koja čuva polja specifična za potklasu. Da bi se došlo do svih polja i nasleđenih i sopstvenih, moraju se raditi spajanja nad tabelama.
- **Apstraktna bazna klasa** – u ovoj varijanti natklasa nije entitet nego samo klase koje u python stilu nasleđuju iz nje (stanje tj. polja i informacije o mapiranju stanja u bazu). Drugim rečima, samo za potklase postoje tabele u bazi i njihov sadržaj je takav kao da su nasleđena polja i relacije definisane direktno nad potklasama. U baznu klasu se dodaje deklaracija:
class Meta:
 abstract = true
- **Proksi klase** – ograničena vrsta nasleđivanja gde se u bazi pamti samo bazna klasa, a proksi klase mogu dodati npr. samo metode

Django ORM – nasleđivanje

- Primer iz aplikacije Talking nasleđivanja sa višestrukim tabelama: model Post i izvedeni modeli VideoPost i ImagePost.
- Bazni model Post se ne menja.
- Model VideoPost izveden je iz modela Post i ima sopstveno polje videoUrl, ostalo nasleđuje. Django automatski ubacuje post_ptr_id strani ključ ujedno i primarni
- Analogno tome se definiše i klasa ImagePost.

```
class VideoPost(Post):
```

```
    videoURL = models.CharField(max_length=255)
```

```
class ImagePost(Post):
```

```
    imageURL = models.CharField(max_length=255)
```

```
CREATE TABLE "djtalk_imagepost" ("post_ptr_id" bigint NOT NULL PRIMARY KEY  
REFERENCES "djtalk_post" ("id") DEFERRABLE INITIALLY DEFERRED,  
"imageURL" varchar(255) NOT NULL);
```

```
CREATE TABLE "djtalk_videopost" ("post_ptr_id" bigint NOT NULL PRIMARY KEY  
REFERENCES "djtalk_post" ("id") DEFERRABLE INITIALLY DEFERRED,  
"videoURL" varchar(255) NOT NULL);
```

Django ORM – nasleđivanje

```
>>> from djangotalk.models import Post, VideoPost, ImagePost, User
>>> post = VideoPost(title='video title', content = 'video desc', videoURL = 'http:',
user=User.objects.get(id=1))
>>> post.save()
>>> post.id
7
>>> post.videopost
<VideoPost: VideoPost object (7)>
>>> videoPost = VideoPost.objects.first()
>>> videoPost.post_ptr
<Post: Post object (7)>
>>> for post in Post.objects.all():
...     print(post.title)
...
Parallel computing
Jurassic Park
China-Made Zhaoxin CPU Hits Retail Market
video title
>>>
```

- Rad sa objektom modela VideoPost. On naravno ima identitet Post objekta i nalazi se u kolekciji tih objekata. Preko reference na Post objekat može se doći do reference na VideoPost (downcast) sa post.videopost, naravno ako se stvarno radi o video postu inače se baca izuzetak
- Video post ima sve svoje i nasleđene attribute, a do reference na osnovni post može se doći preko post_ptr atributa.

Django ORM – korišćenje relacija

- Entitet se može učitati preko drugog već dohvaćenog entiteta ako između njih postoji definisana asocijacija (koje se koriste kao “dinamička polja”. Neki primeri iz aplikacije Talking (deo je u kontrolerima, deo u pogledima):

```
>>> users = User.objects.all()
>>> for user in users:
...     for post in user.post_set.all():
...         print(post.title)
```

- Moguće je koristiti i selekciju i filtriranje i na relacijama.
- Podrazumevano, tip dohvatanja za neku relaciju je LAZY, to jest, entitet povezan sa objektom Post učitaće se iz baze pri prvom referisanju preko relacije, posebnim upitom. što potencijalno rezultuje u velikom broju upita kada se npr. iterira kroz sve postove, a onda traže njihovi tagovi. Ako se želi da prilikom učitavanja objekta post odmah budu učitan i sa njim povezani objekti, treba specificovati vrstu dohvatanja **prefetch_related**.
- Za primer Postova i njihovih Tagova: Bez with bilo bi N+1 SQL upit u bazu za N postova, sa with samo 2 upita: 1 nad tabelom posts, a drugi nad tabelom tags.

```
>>> posts = Post.objects.prefetch_related('tag').all()
>>> for post in posts:
...     for tag in post.tag.all():
...         print(tag.label)
```

Django ORM – ažuriranje relacija

Relacija 1:1 i jedinična strana 1:N

- Primer: User i ContactData. Ako želimo da ažuriramo kontakt podatke korisnika, koristimo `save()` nad relacijom. Pažnja, ne `save()` nad objektom korisnika, jer se tako neće zapamtiti izmene u bazi,

```
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'malina@example.com'
>>>
>>> user.contactdata.email = 'visnja@example.com'
>>> user.save()
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'malina@example.com'
>>>
>>> user.contactdata.email = 'visnja@example.com'
>>> user.contactdata.save()
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'visnja@example.com'
```

Django ORM – ažuriranje relacija

Relacija 1:N, strana sa kardinalnošću N i M:N

- Objekat može da se kreira i upiše u bazu sa **create()**
- Za dodavanje objekata na N stranu relacije koristi se **add()**. Objekti treba da već postoje u bazi. Automatski se radi i save relacije.

```
#models.py
```

```
class Blog(models.Model):  
    pass
```

```
class Author(models.Model):  
    name = models.CharField(max_length=255)
```

```
class Entry(models.Model):  
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE, null=True)  
    authors = models.ManyToManyField(Author)
```

```
>>> from djtalk.models import Entry, Author
```

```
>>> entry = Entry.objects.create() # radi i  
kreiranje objekta i save u bazu
```

```
>>> a1 = Author.objects.create(name="Paja")
```

```
>>> a2 = Author.objects.create(name="Zika")
```

```
>>> a3 = Author.objects.create(name="Laza")
```

```
>>> entry.authors.add(a1, a2, a3)
```

Django ORM – ažuriranje relacija

Uklanjanje relacija među objektima modela

- Za uklanjanje svih objekata iz relacije (ali ne i samih objekata iz baze) koristi se **clear()** nad relacijom.
- Za selektivno uklanjanje jednog ili više objekata iz relacije (bez delete samog objekta iz baze) koristi se **remove(o1, o2,..)** nad relacijom.

```
>>> entry.authors.add(a1, a2, a3)
```

```
>>> entry.authors.all()
```

```
<QuerySet [  
<Author: Author object (4)>, <Author: Author object (5)>, <Author: Author object (6)>]>
```

```
>>> entry.authors.remove(a1, a2)
```

```
>>> entry.authors.all()
```

```
<QuerySet [  
<Author: Author object (6)>]>
```

Django ORM – transakciona obrada

- Transakcija je skup operacija nad bazom koje se izvršavaju kao jedna nedeljiva celina (atomično). Dakle ili se sve izvrše (commit), ili se ni jedna ne izvrši (rollback).
- Transakcionom obradom u Django upravljaju dva podešavanje u settings.py vezana za bazu DATABASES = {}. To su AUTOCOMMIT, podrazumevano TRUE i ATOMIC_REQUESTS, podrazumevano FALSE.
- Podrazumevani autocommit znači da je svaki pojedinačni upit u bazu (rezultat raznih save()) operacija upakovan u sopstvenu transakciju.
- Ako se stavi ATOMIC_REQUESTS na TRUE. svaka Django view funkcija koja obrađuje jedan http zahtev, upakovana je u svoju transakciju. Ovo može imati uticaj na performanse sistema.

Django ORM – transakciona obrada

- Django obezbeđuje eksplicitnu kontrolu transakcija korišćenjem konstrukcije `atomic`, kao dekoratora, ili kao kontekst menadžera.
- U oba slučaja transakcija se uspešno završava (`commit`) ako nije bilo bacanja izuzetaka u obuhvaćenom kodu, inače se radi poništavanje (`rollback`).

```
from django.db import transaction
```

```
@transaction.atomic
```

```
def viewfunc1(request):  
    # This code executes inside a transaction.  
    do_stuff()
```

```
def viewfunc2(request):  
    # This code executes in autocommit mode (Django's default).  
    do_stuff()
```

```
with transaction.atomic():  
    # This code executes inside a transaction.  
    do_more_stuff()
```

Jedan problem sa django finderima

```
#models.py
```

```
class Order(models.Model):  
    sum = models.IntegerField(default=0)
```

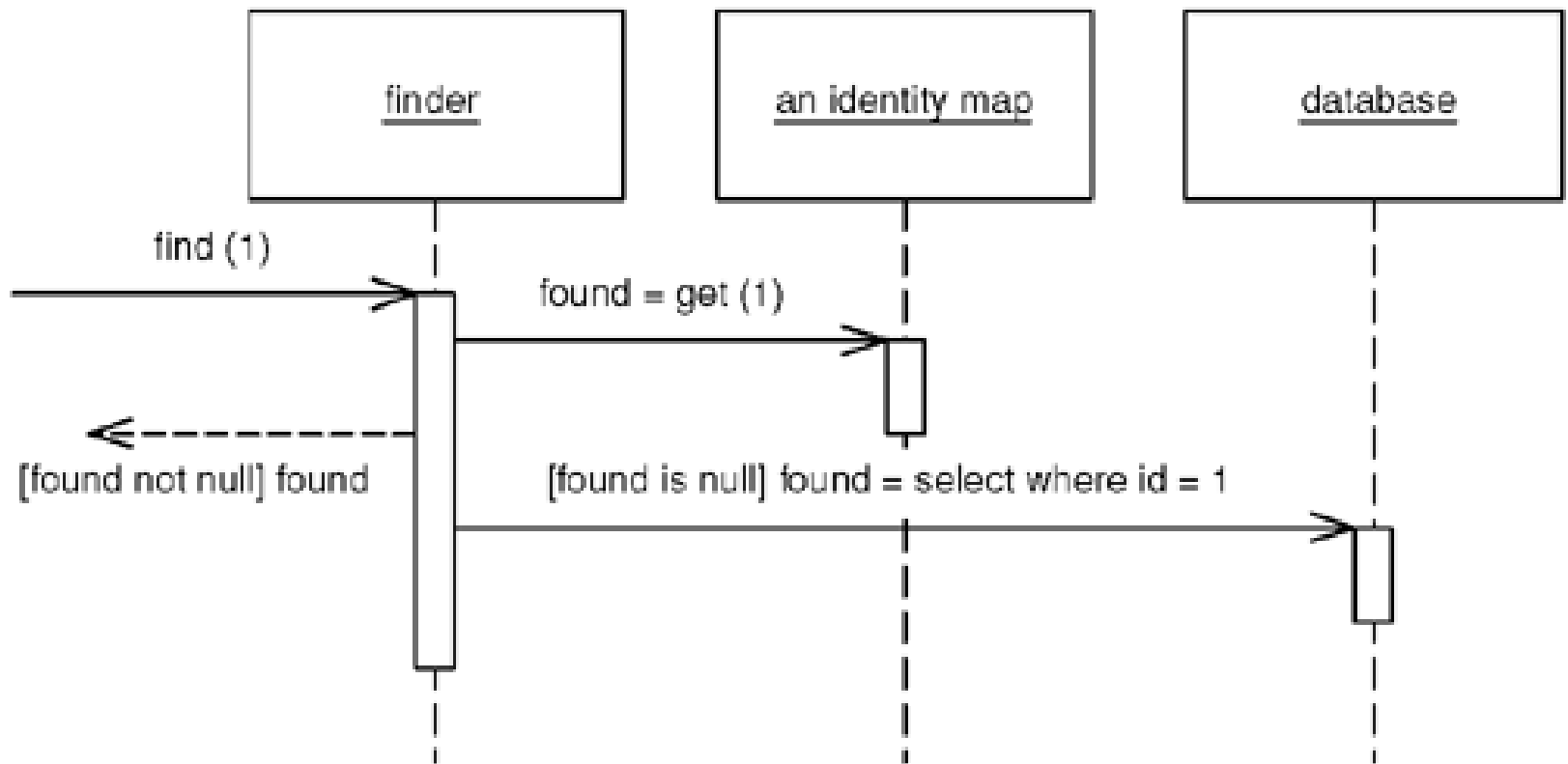
```
>>> from djangotalk.models import Order  
>>> order = Order.objects.create()  
>>> order.sum  
0  
>>> order.sum = 200  
>>> order.id  
2  
>>> order2 = Order.objects.get(id=2)  
>>> order2.sum  
0  
>>> order2.sum = 100  
>>> order.save()  
>>> order2.save()  
>>> order3 = Order.objects.get(id=2)  
>>> order3.sum  
100
```

- Problemi konzistencije podataka nastaju kada možemo napraviti različita dva memorijska objekta za isti entitet u bazi. Rešenje na nivou obrade kompletnog zahteva jednog korisnika je uzorak **Identity Map**.
- Kada se uzme u obzir konkurentni pristup bazi od strane više korisnika, jedno rešenje je eksplicitno koristiti database transakcije u kodu koji koristi domenske objekte.
- Alternativno, može se transakciona obrada ugraditi u Data Mapper kod putem **Unit of Work** patterna.

OBRAZAC IDENTITY MAP (MAPA IDENTITETA)

Obrazac Mapa identiteta

- Obezbeđuje da svaki objekat bude učitani samo jednom držeći svaki učitani objekat u mapi. Traži objekte uz pomoć mape kada se referiše na njih.



Dodavanje Identity Map-a u Django

- Učitava instance modela samo jednom u memoriju prvi put kada zatrebaju
- Obezbeđuje deljenje instance sa istim id-jem svuda u kodu do kraja obrade http zahteva

- Instalacija

```
pip install django-idmap
```

Takodje dodati idmap u settings.py u INSTALLED_APPS

- Osnovno korišćenje:
- Klase modela naslediti iz idmap.models.IdMapModel umesto django.db.models

```
from idmap import models  
class MyModel (models.IdMapModel):
```

```
...
```

**OBRAZAC UNIT OF WORK
(JEDINICA RADA)**

Obrazac Jedinica rada

- Održava listu objekata uključenih u poslovne transakcije i koordinira upisivanje promena i rešava probleme konkurencije.
- Kada vučete podatke iz baze podataka, važno je da pratite šta ste menjali; U suprotnom, podaci neće biti zapisani nazad u bazu. Slično morate da ubacite nove objekte koje ste kreirali i uklonite sve objekte koje ste obrisali.
- Možete da menjate u bazi podataka sa svakom promene vašeg objektnog modela, ali to može da dovede do mnogo malih pristupa bazi, koji se sporo izvršavaju. Dodatno je potrebno da imate otvorenu transakciju tokom cele interakcije, što je nepraktično ako imate poslovnu transakciju koja obuhvata više zahteva.
- Situacija je još gora ako treba da pratite objekte koje ste pročitali, da bi izbegli nedoslednosti.
- Jedinica rada vodi evidenciju o svemu što radite tokom poslovne transakcije što može da utiče na bazu podataka. Kada završite, ona preduzme sve što treba da se uradi da promeni bazu podataka kao rezultat vašeg rada.

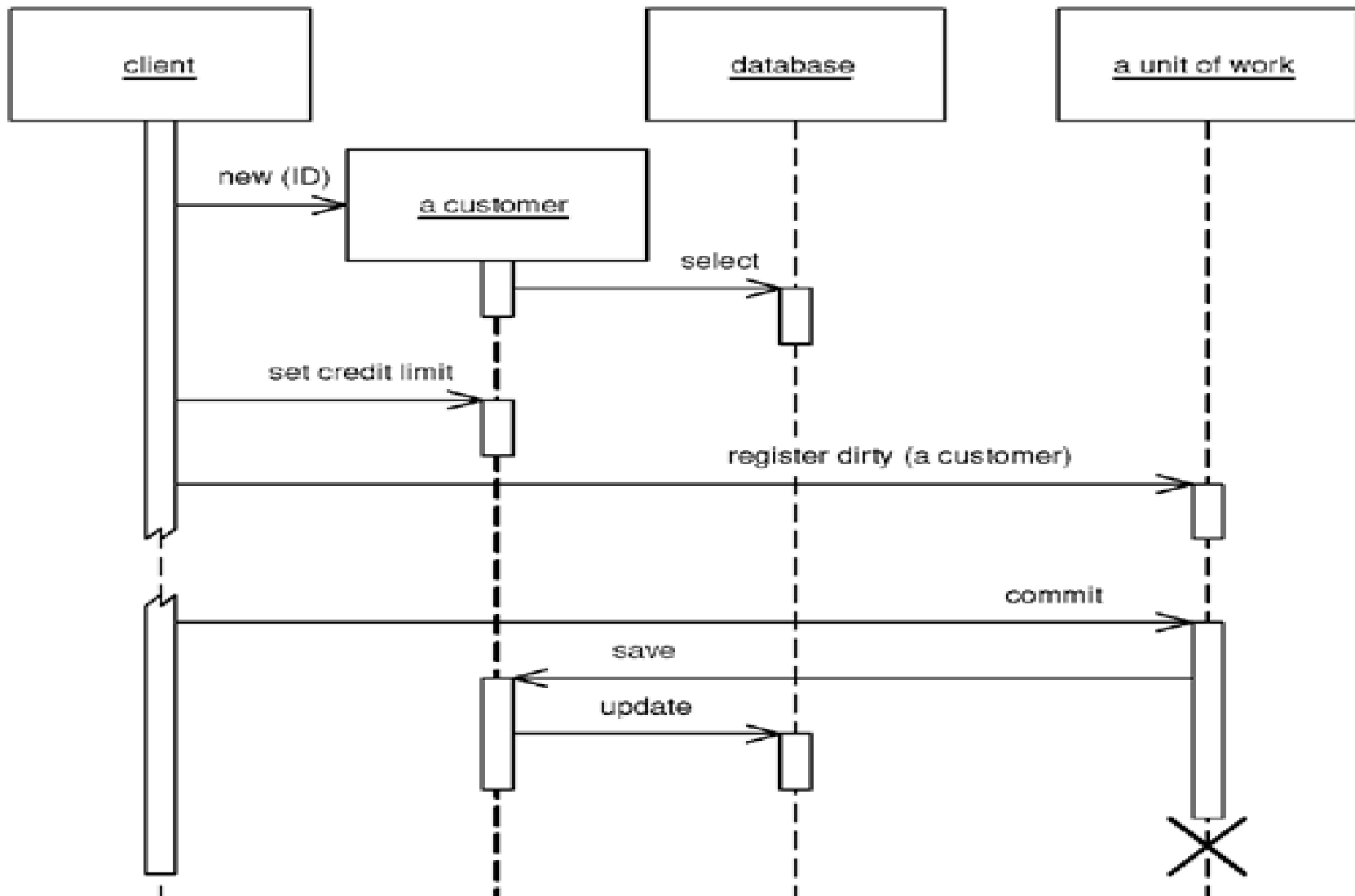
Unit of Work
registerNew(object) registerDirty (object) registerClean(object) registerDeleted(object) commit()

Kako funkcioniše Jedinica rada

- Svaki put kada se kreira, menja ili briše objekat to se saopšti jedinici rada. Ona takođe zna o objektima koji su učitani iz baze tako da može da proveri nekonzistencije u čitanju tako što proverava da nijedan od objekata nije promenjen u bazi podataka u toku poslovne transakcija.
- Ključna stvar o jedinici rada je da, kada dođe vreme da se izvrši commit, jedinica rada odlučuje šta da radi. Ona otvara transakciju, radi kontrolu konkurentnosti (koristi pesimističko ili optimističko zaključavanje), i zapisuje promene u bazi podataka.
- Programer nikada eksplicitno ne poziva metode ažuriranja baze. Na taj način on ne mora da prati šta se promenilo ili da brine o tome kako referencijalni integritet utiče na poredak u kome treba da se rade stvari.
- Programer (ili sam poslovni objekat) mora jedinici rada da saopšti o kojim objektima treba da vodi računa.

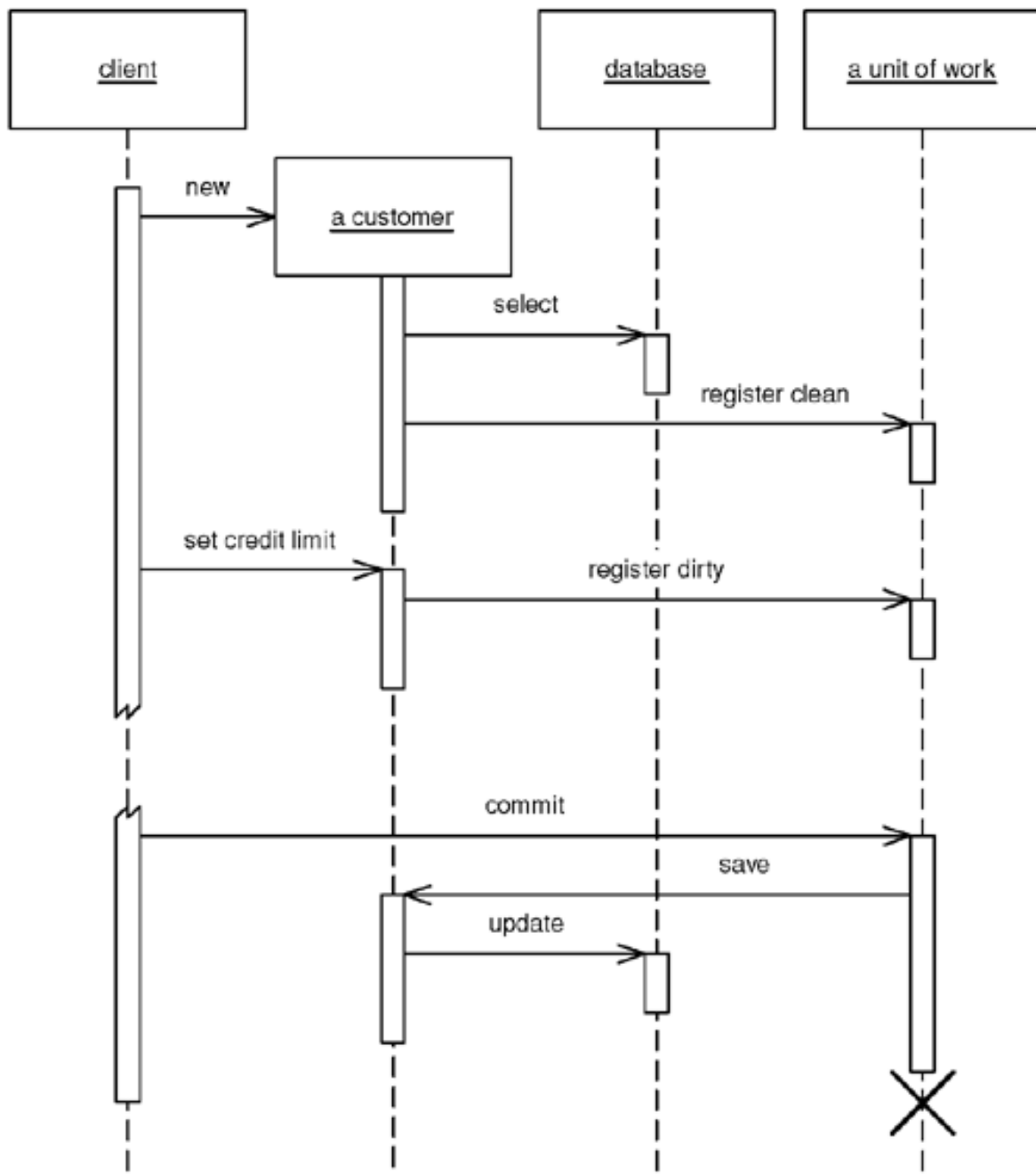
Pozivalac registruje promenjeni objekat

- Sa registracijom od strane pozivaoca, korisnik objekta mora registrovati objekat za promene kod Jedinice rada. Objekat koji nije registrovan neće biti upisan pri commitu.



Objekat sam sebe registruje

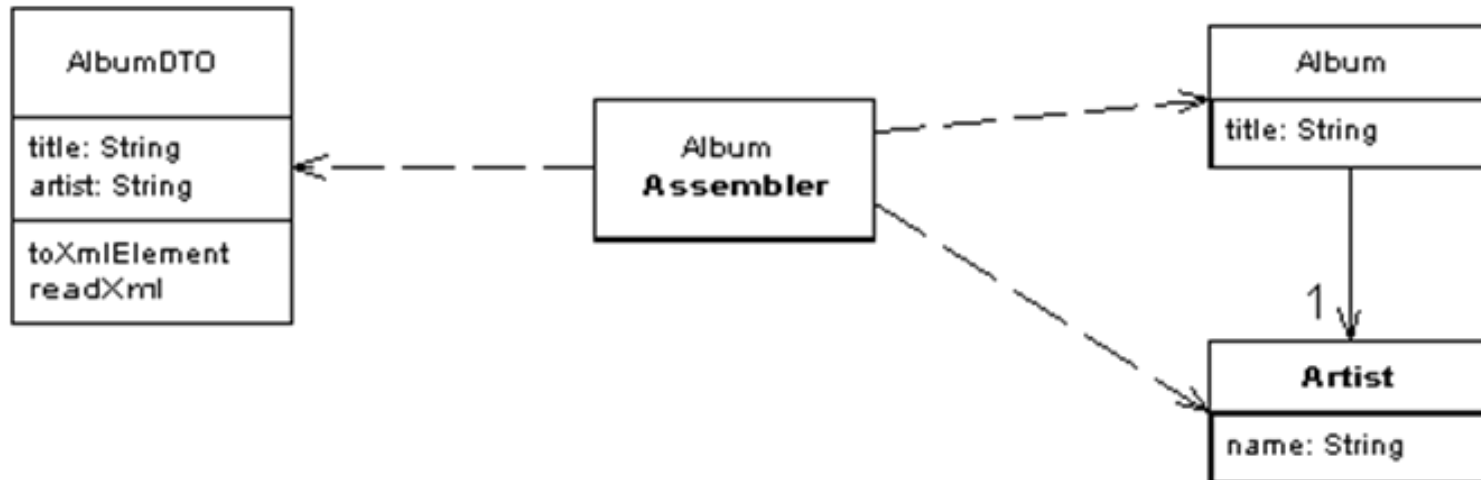
- Registracija se vrši u metodama objekta.
- Učitavanje objekta iz baze registruje objekat kao čist;
- setXY metode registruju objekat kao prljav.
- Jedinica za rad treba da bude prosleđena objektu ili da bude na nekom dobro poznatom mestu.



OBRAZAC DATA TRANSFER OBJECT (OBJEKAT ZA PRENOS PODATAKA)

Obrazac Data Transfer Object

DTO -- Objekat koji prenosi podatke između procesa kako bi se smanjio broj poziva metoda.



- Kada radite sa udaljenim interfejsom, svaki njegov poziv je skup. Kao rezultat toga, potrebno je da smanjite broj poziva, a to znači da morate da prenesete više podataka sa svakim pozivom.
- Rešenje je kreiranje **objekta za prenos podataka (Data Transfer Object)** koji može da sadrži sve podatke za poziv. Mora biti serijalizabilan da bi se prenosio preko veze. Obično se **assembler** koristi na strani servera za prenos podataka između DTO-a (DTO) i bilo kojih objekata domena.

Data Transfer Object

- Kad god udaljenom objektu trebaju neki podaci, on traži odgovarajući objekat za prenos podataka. Objekat za prenos podataka obično nosi mnogo više podataka nego što je udaljeni objekat zahtevao, ali bi trebalo da nosi sve podatke koji će udaljenom objektu biti potrebni neko vreme. Zbog troškova latencije udaljenih poziva, bolje je pogrešiti na strani slanja previše podataka nego morati da se obavlja više poziva. Jedan objekat za prenos podataka obično sadrži više od samo jednog objekta servera. On agregira podatke iz svih objekata servera od kojih udaljeni objekat verovatno želi podatke.
- Obično ne možete prenositi objekte iz modela domena. To je zato što su objekti obično povezani u složenoj mreži koju je teško, ako ne i nemoguće, serijalizovati.
- Polja u objektu za prenos podataka su prilično jednostavna, budući da su primitivi, jednostavne klase poput stringova i datuma ili drugi objekti za prenos podataka. Bilo koja struktura između objekata za prenos podataka treba da bude jednostavna grafička struktura — obično hijerarhija — za razliku od složenijih grafičkih struktura koje vidite u modelu domena.

DRF Serijalizatori kao primer DTO

DRF serializeri zapravo "Pametni DTO-ovi" jer u sebi objedinjuju DTO, Assembler (konverziju) i Validator.

```
from rest_framework import serializers
```

```
# 1. Domenski objekat (Obična Python klasa -  
nije Django Model)
```

```
class Poruka:
```

```
    def __init__(self, ime, email, poruka):  
        self.ime = ime  
        self.email = email  
        self.poruka = poruka
```

```
# 2. Serializer (DTO + Assembler + Validator)
```

```
class KontaktSerializer(serializers.Serializer):
```

```
    # --- DTO: Definisanje šeme podataka ---  
    ime = serializers.CharField(max_length=100)  
    email = serializers.EmailField()
```

```
    poruka = serializers.CharField()
```

```
# Validator: Poslovna logika pre sklapanja
```

```
def validate_email(self, value):  
    if "spam.com" in value:  
        raise serializers.ValidationError("Domen  
je na crnoj listi.")  
    return value
```

```
# Assembler: mapiranje podataka u objekat
```

```
def create(self, validated_data):  
    # Pretvara JSON data u instancu klase  
    return Poruka(**validated_data)
```

Upotreba za serijalizaciju:

```
moj_objekat = Poruka(ime="Ana", email="ana@primer.com", poruka="Zdravo!")  
serializer = KontaktSerializer(moj_objekat)  
# Ovde se "ispod haube" poziva nasleđeni to_representation()  
print(serializer.data)  
# Rezultat: {'ime': 'Ana', 'email': 'ana@primer.com', 'poruka': 'Zdravo!'}
```

DRF ModelSerializer

Šta **ModelSerializer** dodaje na osnovni **Serializer**?

On dodaje sloj automatizacije (Introspection):

1. **Automatski Field Factory:** Pogleda model, vidi IntegerField i sam napravi odgovarajući serializers.IntegerField.
2. **Ugrađen Assembler:** Zna kako da raspakuje validated_data direktno u model.save().
3. **Unique skraćivanje:** Ako model kaže da email mora biti unikatan, ModelSerializer će sam dodati UniqueValidator bez da ti to pišeš.

Primer iz aplikacije Talking

```
# models.py
```

```
class Post(models.Model):
```

```
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    category = models.ForeignKey(Category, on_delete=models.CASCADE, null=True, blank=True)
    tag = models.ManyToManyField(Tag, blank=True)
    title = models.CharField(max_length=255)
    content = models.TextField()
```

```
# serializers.py
```

```
class PostSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        model = Post
        fields = '__all__'
```

```
# views.py
```

```
class PostViewSet(viewsets.ModelViewSet):
```

```
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

```
# urls.py
```

```
router = DefaultRouter()
router.register(r'posts', PostViewSet)
```

```
urlpatterns = [
    path('api/', include(router.urls))
```

```
]
```

ModelViewSet je najmoćniji alat za API jer automatski obezbeđuje kompletan CRUD (Create, Read, Update, Delete).

Kada dr poveže sa **Router**-om, on sam pravi rute za:

list() – Prikaz svih (GET /api/posts/)

create() – Pravljenje nove (POST /api/posts/)

retrieve() – Prikaz jedne (GET /api/posts/id/)

update() – Izmena (PUT /api/posts/id/)

destroy() – Brisanje (DELETE /api/posts/id/)

Fokus: Podaci (JSON).

Zaključak: Put podatka kroz obrasce

Podatak u modernom web sistemu nije samo statični zapis, već entitet koji menja formu kroz različite slojeve arhitekture:

- 1. U bazi podataka:** On je relacioni zapis, ali zahvaljujući **Active Record** obrascu (Django ORM), on oživljava kao Python objekat koji „zna” kako da se sačuva ili obriše.
- 2. U memoriji (Sloj perzistencije):** Django koristi **Unit of Work** i **Identity Map** (kroz QuerySet i keširanje objekata unutar jednog requesta) kako bi osigurao da ne radimo suviše upite i da uvek imamo konzistentnu sliku istog podatka.
- 3. U konstrukciji upita: Query Builder** nam omogućava da deklarativno i precizno „naručimo” podatke od baze bez pisanja sirovog SQL-a, prilagođavajući se potrebama aplikacije u realnom vremenu.
- 4. Na granici sistema (API):** Ovde nastupa **DTO** (Serializer). On odvaja unutrašnju strukturu baze od onoga što svet vidi. Uz pomoć **Assembler**, on transformiše kompleksne modele u jednostavne formate (JSON), dok **Validator** garantuje da u naš sistem ulaze samo ispravni podaci.

Zaključak

- **Ključna poruka:** Ovi obrasci nam omogućavaju **razdvajanje odgovornosti (Separation of Concerns)**.
- Promena u bazi ne mora da sruši frontend, a promena u prikazu ne mora da menja bazu.
- Arhitektura je stabilna onoliko koliko su ove granice jasno definisane.

Literatura

- Martin Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley 2002.
- Django documentation, <https://docs.djangoproject.com/en/>