



Rīgas Tehniskā universitāte
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts

PostgreSQL un PostGIS telpisko datubāzu vadības sistēmas

Izstrādāja: Reinis Veips
Stud.ap.nr. 081RDB032
RDGDB 1.grupa

Rīga – 2013

Satura rādītājs

Par PostgreSQL.....	2
Īsa PostgreSQL vēsture.....	2
POSTGRES projekts.....	2
Postgres95.....	2
PostgreSQL.....	3
PostgreSQL ģeometrisku datu glabāšanai.....	4
Ģeometrisku datu tipi.....	4
Punkts.....	4
Nogrieznis.....	4
Taisnstūri.....	4
Daudzstūri.....	5
Poligons.....	5
Riņķa līnija.....	5
Datu shēmas izveidošanas, datu ievades piemērs.....	6
Funkcijas un operatori darbam ar ģeometrijas datu tipiem.....	8
PostGIS.....	10
Koordinātu sistēmas.....	10
Ģeometrijas datu tips.....	10
Ģeogrāfiskais datu tips.....	11
Datubāzes izveidošana.....	12
Metadatu glabāšana.....	15
Indeksi.....	16
Django un GeoDjango.....	18
GeoDjango.....	18
Django instalācija.....	18
Demonstrācijas lietojuma izveide.....	18
Objektu relāciju attēlojuma slānis.....	28
Datu vizualizācija ar GeoDjango.....	30
OpenLayers.....	30
Secinājumi.....	34
Izmantotā literatūra.....	35

Par PostgreSQL

PostgreSQL ir objektu-relāciju datubāzu vadības sistēma. Tā ir bāzēta uz POSTGRES- Kalifornijas universitātes Berklijas datorzinātņu fakultātes izstrādātnēm. Saskaņā ar PostgreSQL dokumentāciju, POSTGRES projekta ietvaros Kalifornijas universitātes mācībspēki pētīja un ieviesa vairākus konceptus relāciju datubāzēs, kuri tikai vēlāk kļuva pieejami komerciālās datubāzu sistēmās.

PostgreSQL ir atvērtā pirmkoda pēctecis oriģinālajai POSTGRES sistēmai. Tā atbalsta lielu SQL standarta daļu, un piedāvā arī modernas iespējas- kā galveno varu atzīmēt ko līdzīgu versiju kontrolei objektiem datubāzē.

Lietotāji var paplašināt PostgreSQL vairākos veidos:

- Savi datu tipi,
- Funkcijas
- Operatori,
- Agregātfunkcijas,
- Indeksēšanas metodes,
- Procedurālas valodas, kas darbojas datubāzes ietvaros.

Īsa PostgreSQL vēsture

Objektu-relāciju datubāzu vadības sistēma, kas tagad pazīstama ar nosaukumu PostgreSQL ir atvasināta no POSTGRES, kas tika izstrādāta Kalifornijas universitātē, Berklijā. Vairāk kā 20 gadu izstrādes laikā, PostgreSQL ir iespējām bagātākā atvērtā pirmkoda datubāzu vadības sistēma, kas pašlaik pieejama.

POSTGRES projekts

POSTGRES projektu, kuru vadīja profesors *Michael Stonebraker*, sponsorēja vairākas ASV organizācijas: *Defense Advanced Research Projects Agency* (DARPA), Armijas izpētes centrs (ARO) Nacionālais Zinātnes fonds un citi. POSTGRES realizācija sākās 1986. gadā.

POSTGRES sistēma ir tikusi izmantota, lai radītu vairākus izpētes projektus, kā arī lietojumus, kas izmantoti reālā vidē. Piemēri: finansu datu analīzes sistēma, reaktīvā dzinēja veiktspējas novērtēšanas sistēma, asteroīdu izsekošanas datubāze, medicīnas informācijas datubāze, kā arī vairākās ģeogrāfiskās informācijas sistēmas.

Postgres95

1994. gadā, *Andrew Yu* un *Jolly Chen* izveidoja un pievienoja SQL valodas interpretatoru POSTGRES kodam. Rezultātu publicēja kā atvērtā pirmkoda datubāzu vadības rīku, Postgres95. Šajā pakotnē tika izlabotas daudzas problēmas- gan kļūdas programmatūras kodā, ātrdarbības problēmas. Tika uzlabota dokumentācija, un uzrakstītas bibliotēkas piekļuvei pie datubāzes no ārējām programmām.

Pēc šīm izmaiņām Postgres95 piedzīvoja lietotāju skaita trīskāršošanos, taču bija jomas, kurās Postgres95 atpalika no tajā laikā pieejamām komerciālām sistēmām.

PostgreSQL

1996. gada beigās, divus gadus pēc Postgres95 izlaišanas, kļuva skaidrs, ka Postgres95 nosaukums neizturēs laika pārbaudi. Tika izvēlēts jauns nosaukums- PostgreSQL. Nosaukumā ir atsauce uz sākotnējo, POSTGRES projektu, un piedēklis SQL, lai paziņotu par SQL atbalstu.

Arī mūsdienās, neformālā vidē PostgreSQL tiek dēvēts par Postgres- vai nu dēļ tradīcijām, vai dēļ vieglākas izrunāšanas.

Postgres95 izstrādes laikā galvenā uzmanība tika veltīta, lai identificētu un saprastu problēmas esošajā servera puses kodā. Sākot ar PostgreSQL, lielāka uzmanība ir tikusi veltīta, lai paplašinātu datubāzes iespējas, taču darbs turpinās arī citās jomās.

PostgreSQL ģeometrisku datu glabāšanai

Lai gan PostgreSQL iebūvētie datu tipi pēc autora uzskatiem nav pilnīgi piemēroti ģeogrāfiskās informācijas sistēmām, PostgreSQL piedāvā 2-dimensiju ģeometrijas datu tipus un metodes ģeometrisku aprēķinu veikšanai.

Pārskatot PostgreSQL dokumentāciju, pieejamie datu tipi varētu būt piemēroti vienkāršu 2-dimensionālu datu glabāšanai un apstrādei. Šajā kontekstā ar 2-dimensionāliem datiem tiek saprasti ģeometriski objekti Dekarta koordinātu sistēmā (X un Y asis, kas atrodas vienā plaknē un ir perpendikulāras viena otrai).

Ģeometrisku datu tipi

PostgreSQL ir iebūvēti sekojoši datu tipi:

- punkts,
- taisne,
- nogrieznis,
- taisnstūris,
- 3 datu tipi dažādu daudzstūru aprakstīšanai,
- riņķa līnija.

PostgreSQL piedāvā arī iespēju veikt operācijas ar šiem datiem (mērogot, pārvietot, rotēt, noteikt šķēlumus u.c.).

Punkts

Punkts (point) ir pamata datu vienība ģeometriskiem tiem. Punkta definēšanai tiek izmantoti sekojoši sintakses varianti:

```
( x , y )  
x , y
```

kur x un y ir decimāldaļskaitļi, kas apzīmē punkta koordinātas.

Nogrieznis

Nogriežņi (lseg) tiek apzīmēti kā punktu pāri. Nogriežņa definēšanai var tikt izmantota sekojoša sintakse:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]  
( ( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

kur (x1, y1) un (x2, y2) ir nogriežņa sākuma un beigu punkti.

Taisnstūri

Taisnstūri (box) tiek apzīmēti ar koordinātu pāri, kur katra koordināta atrodas taisnstūra diagonāli pretējos stūros.

Definēšanai izmanto sekojošos sintakses variantus:

```
(( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

kur (x1, y1) un (x2, y2) ir jebkuri diagonāli pretējie taisnstūra stūri.

Ievadē var izmantot jebkurus taisnstūra diagonāli pretējos stūrus, taču vērtības datubāzē tiks pārkārtotas, lai glabātu šāda secībā: labais augšējais un kreisais apakšējais punkts.

Daudzstūri

Daudzstūri (*path*) tiek apzīmēti kā saraksts ar savienotiem punktiem. Daudzstūris var būt noslēgts, ja pēdējais punkts ir savienots ar pirmo, vai arī atvērts- ja pēdējais punkts nav savienots ar pirmo punktu.

Daudzstūru definēšanai izmanto sekojošu sintaksi:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]  
( ( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1 , ... , xn , yn )  
x1 , y1 , ... , xn , yn
```

Kvadrātiekvādas izmanto atvērta daudzstūra apzīmēšanai, savukārt parastas iekavas- aizvērta daudzstūra apzīmēšanai.

Ja definīcijā neizmanto iekavas, daudzstūris tiek uzskatīts par noslēgtu.

Poligons

Poligoni (*polygon*) ir ļoti līdzīgi daudzstūrim, taču tie obligāti ir noslēgti. Datubāzes iekšienē poligoni tiek glabāti savādāk, un ir plašākas iespējas operēt ar poligoniem.

Poligonu vērtības var definēt sekojoši:

```
(( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1 , ... , xn , yn )  
x1 , y1 , ... , xn , yn
```

Jāatzīmē, ka tā kā daži sintakses varianti ir identiski daudzstūra definīcijai, svarīgs ir konteksts ievadot šos datu tipus (kolonna, kurā tie tiek glabāti, vai procedūras/funkcijas argumenta tips).

Riņķa līnija

Riņķa līnijas (*circle*) apzīmē ar riņķa līnijas sākumpunktu un rādiusu.

Var izmantot sekojošu sintaksi:

```
< ( x , y ) , r >  
( ( x , y ) , r )  
( x , y ) , r  
x , y , r
```

kur (x, y) ir sākumpunkts, un r- riņķa līnijas rādiuss.

Datu shēmas izveidošanas, datu ievades piemērs

Lai iepazītos ar ģeometrijas datu tiptiem, autors izvēlējās problēmu- aprakstīt datubāzē telpu plānus, ievietot datus par dažām telpām, un mēģināt izgūt datus (teiksim, lielākā telpa ēkā).

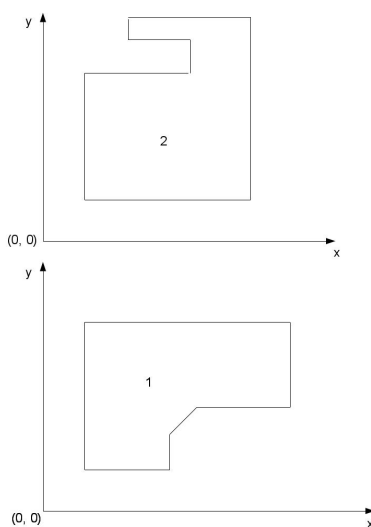
Darbībā ar datubāzi tika izmantots psql komandrindas klients. Šis komandrindas klients ļauj vērsties pie datubāzes ar SQL vaicājumiem, kā arī izmantojot šī rīka komandas (rindas, kas sākas ar \)- iegūt datus par datubāzi.

```
rtu_postgres=# CREATE TABLE telpu_plani (  
rtu_postgres=# id SERIAL,  
rtu_postgres=# telpa_pol polygon)  
rtu_postgres=# ;  
CREATE TABLE  
rtu_postgres=# \dt  
List of relations  
Schema | Name | Type | Owner  
-----+-----+-----+-----  
public | telpu_plani | table | reinis  
(1 row)  
  
rtu_postgres=# \d+ telpu_plani  
Table "public.telpu_plani"  
Column | Type | Modifiers | Storage |  
Stats target | Description  
-----+-----+-----+-----+-----  
id | integer | not null default nextval('telpu_plani_id_seq'::regclass) | plain |  
|  
telpa_pol | polygon | | extended |  
|  
Has OIDs: no
```

Tika izveidota tabula ar divām kolonnām- id (automātiski palielinās ievietojot jaunus ierakstus) un kolonnu telpa_pol, ar datu tipu- poligons.

Tālāk šajā tabulā tiek ievietoti dati. Datu grafisks attēlojums ir redzams zemāk (zīmēts pirms datu ievadīšanas, lai izrēķinātu koordinātas).

Pirms datu ievadīšanas tika izvēlēts mērogs- 1 vienība atbilst 1 metram.



Attēls 1: Shematisks ievietojamo datu attēlojums

```

rtu_postgres=# INSERT INTO telpu_plani (telpa_pol) VALUES ('( (2, 2), (2, 9), (12, 9), (12, 5), (8, 5), (4, 6), (4, 2))');
INSERT 0 1

rtu_postgres=# INSERT INTO telpu_plani (telpa_pol) VALUES(' (2, 2), (2, 8), (7, 8), (7, 9), (4, 9), (4,10), (10, 10), (10, 2) ');
INSERT 0 1

rtu_postgres=# SELECT * FROM telpu_plani;
 id |          telpa_pol
-----+-----
  1 | ((2,2),(2,9),(12,9),(12,5),(8,5),(4,6),(4,2))
  2 | ((2,2),(2,8),(7,8),(7,9),(4,9),(4,10),(10,10),(10,2))
(2 rows)

rtu_postgres=#

```

Lai pārliecinātos, ka poligoni ir aprakstīti pareizi, tika izsauktas dažas PostgreSQL iebūvētas funkcijas, kas apstrādā ģeometriskus datus. Viena no šīm pamata funkcijām ir *area(object)*, kas aprēķina objekta laukumu.

```

rtu_postgres=# SELECT id, area(path(telpa_pol)) AS laukums FROM telpu_plani;
 id | laukums
-----+-----
  1 |      44
  2 |      57
(2 rows)

rtu_postgres=#

```

Kā interesants aspekts ir jāatzīmē autora pirmais vaicājums, kam vajadzēja izgūt šo pašu informāciju. PostgreSQL dokumentācijā funkcijas *area* arguments ir *object*. Mēģinot izpildīt šādu vaicājumu, tika saņemts kļūdas paziņojums:

```

rtu_postgres=# SELECT id, area(telpa_pol) AS laukums FROM telpu_plani;
ERROR:  function area(polygon) does not exist
LINE 1: SELECT id, area(telpa_pol) AS laukums FROM telpu_plani;
                        ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.

```

Ir redzams, ka PostgreSQL dzinējs nevar izpildīt funkciju, kuras argumenta datu tips ir *polygon*. Meklējot informāciju par šādu problēmu internetā, atradu, ka ar funkciju *path()* ir jāpārveido *polygon* uz datu tipu *path*. To, augstāk esošajā vaicājumā dara funkcija *path()*. Darba autora pieņēmums ir tāds, ka šis ir trūkums PostgreSQL dokumentācijā- šāds komentārs tika pievienots arī PostgreSQL dokumentācijas komentārā.

Funkcijas un operatori darbam ar ģeometrijas datu tiem

Operators	Apraksts	Piemērs
+	Pārvietošana	box '((0,0),(1,1))' + point '(2.0,0)'
-	Pārvietošana	box '((0,0),(1,1))' - point '(2.0,0)'
*	Transformēšana (mērogošana/rotācija)	box '((0,0),(1,1))' * point '(2.0,0)'
/	Transformēšana (mērogošana/rotācija)	box '((0,0),(2,2))' / point '(2.0,0)'
#	Krustošanās punkts	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Punktu skaits poligonā vai daudzstūrī	# '((1,0),(0,1),(-1,0))'
@-@	Perimetrs	@-@ path '((0,0),(1,0))'
@@	Figūras centra punkts	@@ circle '((0,0),10)'
##	Tuvākais otrā operanda punkts pirmajam operandam.	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Attālums starp	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Vai figūras pārklājas (vismaz viens kopīgs punkts)	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	Atrodas pa kreisi no	circle '((0,0),1)' << circle '((5,0),1)'
>>	Atrodas pa labi no	circle '((5,0),1)' >> circle '((0,0),1)'
&<	Neturpinās pa labi no	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Neturpinās pa kreisi no	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	Atrodas zemāk par	box '((0,0),(3,3))' << box '((3,4),(5,5))'
>>	Atrodas augstāk par	box '((3,4),(5,5))' >> box '((0,0),(3,3))'
&<	Neturpinās uz augšu no	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Neturpinās uz leju no	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<^	Ir zem (saskare ir atļauta)	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Ir virs (saskare ir atļauta)	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Krustojas?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Ir horizontāls	?- lseg '((-1,0),(1,0))'
?-	Ir paralēli par X asi	point '(1,0)' ?- point '(0,0)'
?	Ir vertikāls	? lseg '((-1,0),(1,0))'
?	Ir paralēli pa Y asi	point '(0,1)' ? point '(0,0)'
?-	Ir perpendikulāri	lseg '((0,0),(0,1))' ?- lseg '((0,0),(1,0))'
?	Ir paralēli	lseg '((-1,0),(1,0))' ? lseg '((-1,2),(1,2))'
@>	Satur (iekļauj pilnībā)	circle '((0,0),2)' @> point '(1,1)'
<@	Satur (iekļauj daļēji)	point '(1,1)' <@ circle '((0,0),2)'
~=	Tāds pats kā	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Tabula 1: Operatori darbam ar ģeometrijas datu tiem

Kā redams 1. tabulā, PostgreSQL piedāvā plašu spektru operatoru darbam ar ģeometrijas datu tiem. Kā galveno problēmu darba autors var atzīmēt nepilnīgo dokumentāciju par šiem operatoriem- kādus datu tipus tie atbalsta, kādi ir atgrieztie datu tipi u.c.

Funkcija	Rezultāta datu tips	Apraksts	Piemērs
area(object)	double precision	laukums	area(box '((0,0),(1,1)))
center(object)	point	centrs	center(box '((0,0),(1,2)))
diameter(circle)	double precision	Riņķa līnijas diametrs	diameter(circle '((0,0),2.0)')
height(box)	double precision	Figūras augstums	height(box '((0,0),(1,1)))
isclosed(path)	boolean	Vai daudzstūris ir noslēgts	isclosed(path '((0,0),(1,1),(2,0)))
isopen(path)	boolean	Vai daudzstūris ir atvērts	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	Garums/perimetrs	length(path '((-1,0),(1,0)))
npoints(path)	int	Punktu skaits figūrā	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	Punktu skaits figūrā	npoints(polygon '((1,1),(0,0)))
pclose(path)	path	Aizvērt daudzstūri	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	Atvērt daudzstūri	popen(path '((0,0),(1,1),(2,0)))
radius(circle)	double precision	Riņķa līnijas rādiuss	radius(circle '((0,0),2.0)')
width(box)	double precision	Figūras platums	width(box '((0,0),(1,1)))

Tabula 2: Funkcijas darbam ar ģeometriskiem datiem

Pētot šīs funkcijas, darba autoru pārsteidza ģeometrisku operāciju atbalsta trūkums- piemēram, apvienojuma, šķēluma. Šis aspekts varētu būt traucējošs ģeometrisku datu tipu pielietošanai reāliem uzdevumiem.

Funkcija	Rezultāta tips	Apraksts	Piemērs
box(circle)	box	Riņķa līnija -> taisnstūris	box(circle '((0,0),2.0)')
box(point, point)	box	Punkti -> taisnstūris	box(point '(0,0)', point '(1,1)')
box(polygon)	box	Daudzstūris -> taisnstūris	box(polygon '((0,0),(1,1),(2,0)))
circle(box)	circle	Taisnstūris -> riņķa līnija	circle(box '((0,0),(1,1)))
circle(point, double precision)	circle	Punkts, rādiuss -> riņķa līnija	circle(point '(0,0)', 2.0)
circle(polygon)	circle	Daudzstūris -> riņķa līnija	circle(polygon '((0,0),(1,1),(2,0)))
lseg(box)	lseg	Taisnstūra diagonāle -> nogrieznis	lseg(box '((-1,0),(1,0)))
lseg(point, point)	lseg	Punkti -> nogrieznis	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	Daudzstūris -> daudzstūris	path(polygon '((0,0),(1,1),(2,0)))
point(double precision, double precision)	point	Punkta izveide	point(23.4, -44.5)
point(box)	point	Taisnstūra centrs	point(box '((-1,0),(1,0)))
point(circle)	point	Riņķa līnijas centrs	point(circle '((0,0),2.0)')
point(lseg)	point	Nogriežņa viduspunkts	point(lseg '((-1,0),(1,0)))
point(polygon)	point	Daudzstūra centrs	point(polygon '((0,0),(1,1),(2,0)))
polygon(box)	polygon	Taisnstūris -> taisnstūra daudzstūris	polygon(box '((0,0),(1,1)))
polygon(circle)	polygon	Riņķa līnija -> 12 punktu daudzstūris	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	Riņķa līnija -> n punktu daudzstūris	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	Daudzstūris-> daudzstūris	polygon(path '((0,0),(1,1),(2,0)))

Tabula 3: Ģeometrijas datu tipu konvertācijas iespējas

Datu tipu pārveidošanas iespējas šķiet pietiekamas reālu uzdevumu veikšanai.

PostGIS

PostGIS ir trešās puses paplašinājums PostgreSQL objektu-relāciju datubāzu vadības sistēmai. PostGIS ļauj datubāzē glabāt, kā arī analizēt un apstrādāt ģeografiskus informācijas sistēmas objektus.

PostGIS pirmo versiju izlaida 2001. gadā, *Refractions Research* organizācija, kas nodarbojas ar ģeogrāfisko un telpisku informāciju sistēmu izstrādi. Kā viens no motivatoriem izstrādāt un publicēt PostGIS bija tieši atvērta pirmkoda *spatial* datubāzu vadības sistēmu trūkums. Tā kā konkurējošas *spatial* informācijas sistēmas nebija modificējamās pietiekoši šīs organizācijas vajadzībām, tā izstrādāja PostGIS, lai varētu ieviest un pētīt dažādus uzlabojumus *spatial* datubāzēm.

PostGIS iespēju pārskats:

- Ģeometrijas datu tipi (punkts, ceļš, poligons, kā arī šo tipu apvienojumi, kolekcijas)
- Operatori ģeogrāfiskiem mērījumiem (attālums, laukums, garums, perimetrs)
- Operatori darbībām ar datu kopām: apvienojums, šķēlums
- R-tree indeksi ģeometriskiem un ģeogrāfiskiem datu tiptiem.

Koordinātu sistēmas

PostGIS glabā datus divās koordinātu sistēmās (*SRID- spatial reference identifier*). Ir pieejama Dekarta koordinātu sistēma 3 dimensijās- XYZ. Šajā koordinātu sistēmā katram punktam iespējams piesaistīt arī metainformāciju- piemēram, laiku, kad koordināta iegūta, iegūstot koordinātu sistēmu XYZM. Jāpiezīmē, ka šīs informācijas fragments ir pielietojams, lai, piemēram, glabājot datus par poligonu, piesaistītu kādu informācijas daļu katrai poligona šķautnei. Tas nav vienīgais pieejamais metadatu glabāšanas veids- ja ir nepieciešams glabāt punktus, kuriem piesaistīta kāda informācija, to droši var darīt veidojot kolonnu ar tipu punkts, un vairākas kolonnas ar ierastiem datu tiptiem- teksts, skaitļi u.c.

Šajā koordinātu sistēmā ir pieejams plašākais operatoru un apstrādes funkciju klāsts, jo tā ir bijusi pieejama PostGIS no pirmsākumiem.

Otra pieejamā koordinātu sistēma, kas izmantojama tieši ģeogrāfiskiem datiem ir WGS 84 garuma/platuma grādi. Šī ir sfēriska koordinātu sistēma, un PostGIS ir ieviesta salīdzinoši nesen, tāpēc tai vēl nav tik plašs operatoru un funkciju atbalsts. Operācijas šajā koordinātu sistēmā ir lēnākas- tas saistīts ar papildu matemātiskajiem aprēķiniem dēļ sfēriskās koordinātu sistēmas.

Ģeometrijas datu tips

PostGIS atbalsta OpenGIS konsorcijs izstrādātās “Vienkāršas iespējas priekš SQL” specifikācijas datu tipus.

Tie būtu:

- POINT(0 0)
- LINESTRING(0 0,1 1,1 2)
- POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2,1 1))

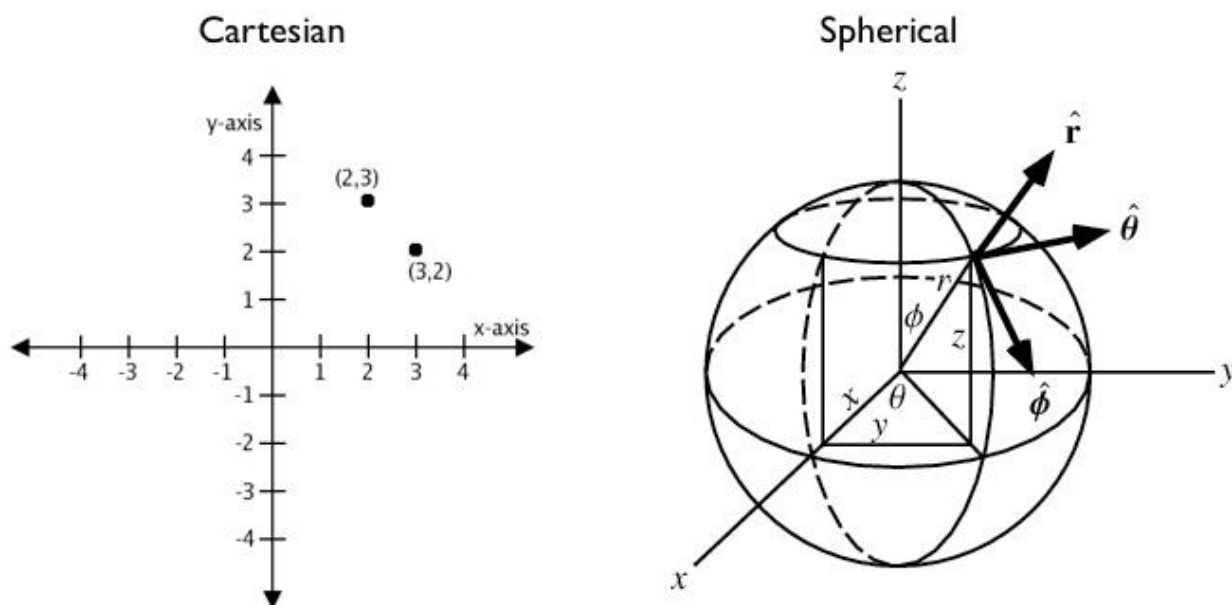
- MULTIPOINT(0 0,1 2)
- MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))
- MULTIPOLYGON((((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))
- GEOMETRYCOLLECTION(POINT(2 3),LINESTRING(2 3,3 4))
- CIRCULARSTRING(0 0, 1 1, 1 0)
- COMPOUNDCURVE(CIRCULARSTRING(0 0, 1 1, 1 0),(1 0, 0 1))
- CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1))
- MULTICURVE((0 0, 5 5),CIRCULARSTRING(4 0, 4 4, 8 4))
- MULTISURFACE(CURVEPOLYGON(CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),(1 1, 3 3, 3 1, 1 1)),((10 10, 14 12, 11 10, 10 10),(11 11, 11.5 11, 11 11.5, 11 11)))

Ģeogrāfiskais datu tips

Ģeogrāfiskais datu tips ir paredzēts datu glabāšanai kartogrāfiskās koordinātās, izmantojot sfērisku koordinātu sistēmu (pretstatā ģeometrijas datu tipam, kas izmanto Dekarta koordinātu sistēmu plaknē). Ģeogrāfiskajā datu tipā, koordinātas tiek glabātas leņķiskajās vienībās- garuma un platuma grādos.

Ģeometrijas datu tipa pamats ir plakne. Tuvākais attālums starp punktiem uz plaknes ir taisna līnija, kas nozīmē to, ka aprēķini ar ģeometrijas datu tipiem var tikt veikti izmantojot relatīvi vienkāršas matemātiskas darbības Dekarta koordinātu sistēmā.

Ģeogrāfijas datu tipa pamats ir sfēra. Īsākais ceļš starp diviem punktiem uz sfēras virsmas ir loks. Tas, savukārt nozīmē, ka aprēķini ar ģeogrāfiskiem datiem ir daudz sarežģītāki un aizņem vairāk laika- arī to realizācijai. Pašlaik PostGIS atbalsta krietni mazāk darbību ar ģeogrāfijas datu tipiem, tāpēc PostGIS izstrādātāji iesaka rūpīgi izvērtēt, vai datu glabāšanai izmantot ģeogrāfijas vai ģeometrijas datu tipu.



Attēls 2: Dekarta un sfēriskās koordinātu sistēmas

Datubāzes izveidošana

Tā kā PostGIS ir PostgreSQL paplašinājums, lielāko daļu darbību ar PostGIS var veikt arī ar PostgreSQL klientiem. Sekojošā piemērā tiek izmantots *psql* komandrindas klients.

Vispirms tiek izveidota datubāze ar nosaukumu *rtu_postgis*, un ar otro komandu tiek palaists *psql* rīks. Komandas arguments ir datubāzes, kuru izmantot.

```
$ ~ createdb rtu_postgis
$ ~ psql rtu_postgis
psql (9.3.2)
Type "help" for help.

rtu_postgis=# CREATE EXTENSION postgis;
CREATE EXTENSION
rtu_postgis=#
```

Ar SQL komandu `CREATE EXTENSION` (PostgreSQL specifisks SQL valodas paplašinājums) tiek iespējots *postgis* paplašinājums pašreiz izvēlētajai datubāzei.

Lai pārliecinātos, ka paplašinājums ir veiksmīgi iespējots, pārbaudīsim paplašinājuma versiju:

```
rtu_postgis=# SELECT postgis_full_version();
               postgis_full_version
-----
POSTGIS="2.1.1 r12113" GEOS="3.4.2-CAPI-1.8.2 r3921" PROJ="Rel. 4.8.0, 6 March 2012"
GDAL="GDAL 1.10.0, released 2013/04/24" LIBXML="2.9.1" LIBJSON="UNKNOWN" RASTER
(1 row)

rtu_postgis=#
```

Kā redzams, PostGIS versija tiek veiksmīgi parādīta, kas nozīmē, ka paplašinājums datubāzē darbojas veiksmīgi.

Izveidosim vienkāršu tabulu, kurā var glabāt ģeometriskus objektus:

```
rtu_postgis=# CREATE TABLE geometries (name varchar, geom geometry);
CREATE TABLE
rtu_postgis=#
```

Ievietosim dažus vienkāršus objektus:

```
rtu_postgis=# INSERT INTO geometries VALUES
('punkts', 'POINT(-2 -2)'),
('cels', 'LINESTRING(0 0, 100 100, 110 10, 0 0)'),
('poligons', 'POLYGON((0 0, 10 8, 6 5, 0 10))'),
('poligons_caurums', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1))'),
('trissturis', 'POLYGON((30 30, 40 40, 40 20, 30 30))');
INSERT 0 5
rtu_postgis=#
```

Pēc objektu ievietošanas, ir jāatjauno metadati. To var darīt manuāli, katrai kolonnai, bet var arī izmantot PostGIS funkciju, kas pārbauda un atjauno metadatu informāciju.

Jāpiezīmē, ka pēc noklusējuma šī funkcija izveido ierobežojumu- katrā geometry tipa kolonnā drīkstētu glabāt tikai vienu apakštipu. Šādu ierobežojumu izveidošanu var atslēgt, nododot funkcijai argumentu false¹:

```
rtu_postgis=# SELECT Populate_Geometry_Columns(false);
WARNING: Could not add geometry type check (POINT) to table column: public.geometries.geom
CONTEXT: PL/pgSQL function populate_geometry_columns(boolean) line 51 at assignment
populate_geometry_columns
-----
probed:1 inserted:1
(1 row)
rtu_postgis=#
```

Mēģinot apskatīt tabulas datus izmantojot acīmredzamu metodi, negūsim saprotamu rezultātu:

```
rtu_postgis=# SELECT * FROM geometries;
 name | geom
-----+-----
punkts | 01010000000000000000000000000000C000000000000000C0
cels   | 0102000000400000000000000000000000000000000000000000000000000000...
poligons | 0103000000010000000040000000000000000000000000000000000000000000...
poligons_caurums | 0103000000020000000050000000000000000000000000000000000000000000...
trissturis | 010300000001000000004000000000000000000000003E400000000000003E4000000000...
(5 rows)
rtu_postgis=#
```

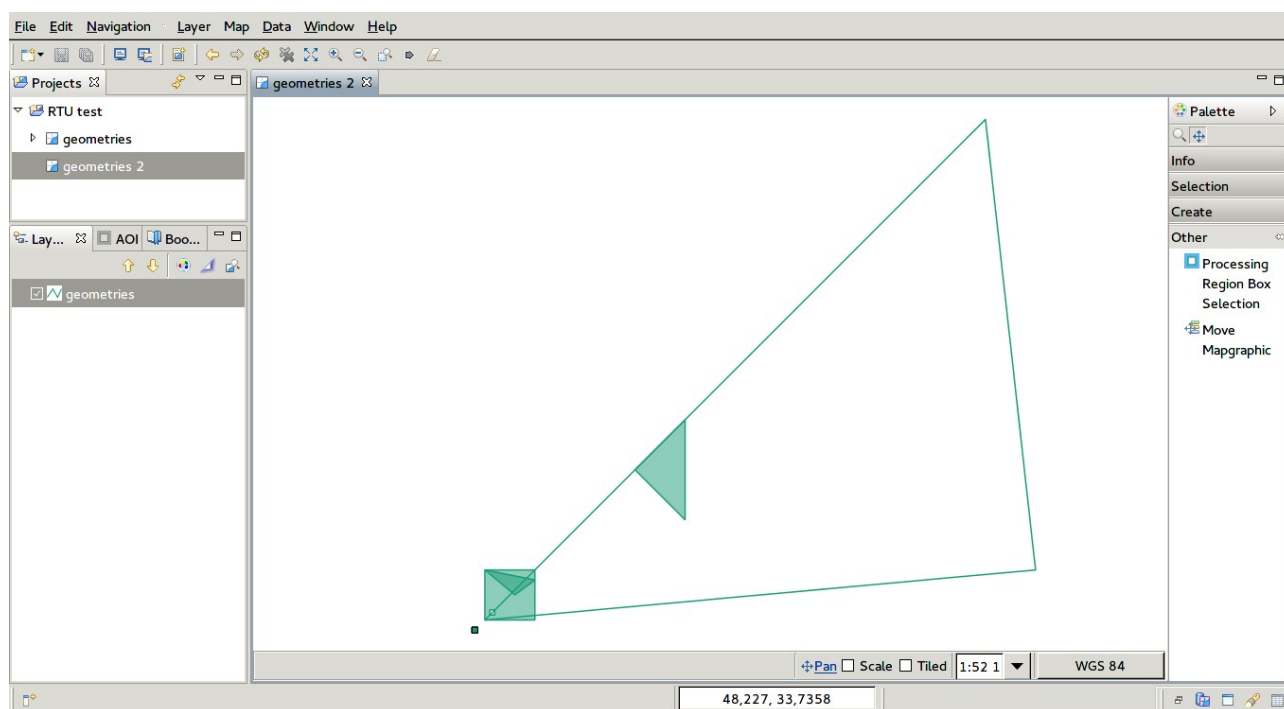
1 http://postgis.net/docs/manual-2.1/Populate_Geometry_Columns.html

Šāds rezultāts tika iegūts, jo dati ģeometrijas kolonnā glabājas binārā formātā. Lai iegūtu pārskatāmāku saturu, jāizmanto ST_AsText() funkcija, kas attēlos ieraksta saturu lasītājam uztveramākā formā:

```
rtu_postgis=# SELECT name, ST_AsText(geom) FROM geometries;
name | st_astext
-----+-----
punkts | POINT(-2 -2)
cels | LINESTRING(0 0,100 100,110 10,0 0)
poligons | POLYGON((0 10,10 8,6 5,0 10))
poligons_caurums | POLYGON((0 0,10 0,10 10,0 10,0 0),(1 1,1 2,2 2,2 1,1 1))
trissturis | POLYGON((30 30,40 40,40 20,30 30))
(5 rows)
rtu_postgis=#
```

Arī šāds formāts nav īsti viegli uztverams, tāpēc eksistē vairāki trešās puses rīki, kas ļauj vizualizēt tabulas saturu grafiskā veidā. Tā kā lielākā daļa no tiem ir paredzēti ĢIS pielietojumiem, dažiem no tiem pietrūkst iespēju filtrēt datus (taču to var apiet, izveidojot skatu vai pagaidu tabulu ar vajadzīgajiem datiem, un vizualizācijas rīkam norādīt, lai tas izmanto datus no skata/pagaidu tabulas).

Un visbeidzot, lai pārliecinātos, ka ģeometrijas dati ir ievietoti korekti, izmantosim trešās puses ĢIS vizualizācijas rīku- uDig. To ir izstrādājusi tā pati organizācija, kas izstrādāja PostGIS.



Attēls 3: uDig interfeiss attēlojot geometries tabulas saturu

Metadatu glabāšana

PostGIS metadatus par ģeometrijas un ģeogrāfijas kolonnām glabā atsvišķos sistēmas skatos.

PostGIS izveido arī dažus citus sistēmas skatus- rastra datiem, kā arī informācijai par izmantotajām telpiskajām koordinātu sistēmām (treknrakstā)

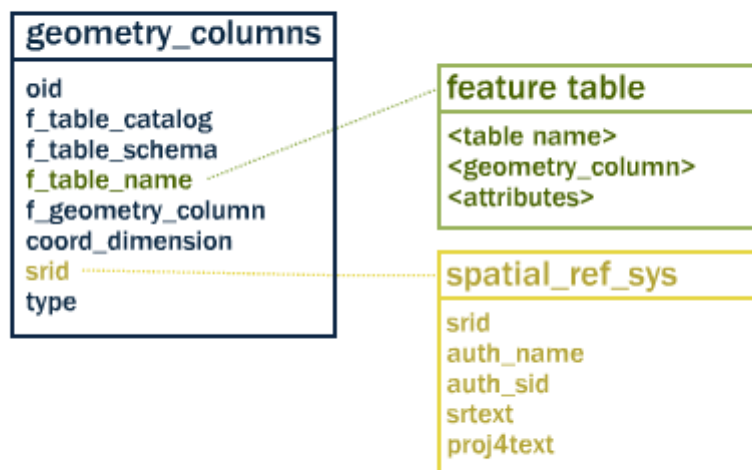
```
$ psql rtu_postgis
psql (9.3.2)
Type "help" for help.

rtu_postgis=# \d
                                List of relations
 Schema |          Name          | Type  | Owner
-----+-----+-----+-----
 public | auth_group             | table | reinis
 public | auth_group_id_seq      | sequence | reinis
 public | auth_group_permissions | table | reinis
 public | auth_group_permissions_id_seq | sequence | reinis
 public | auth_permission        | table | reinis
 public | auth_permission_id_seq | sequence | reinis
 public | auth_user              | table | reinis
 public | auth_user_groups       | table | reinis
 public | auth_user_groups_id_seq | sequence | reinis
 public | auth_user_id_seq       | sequence | reinis
 public | auth_user_user_permissions | table | reinis
 public | auth_user_user_permissions_id_seq | sequence | reinis
 public | django_admin_log       | table | reinis
 public | django_admin_log_id_seq | sequence | reinis
 public | django_content_type    | table | reinis
 public | django_content_type_id_seq | sequence | reinis
 public | django_session         | table | reinis
 public | geography_columns    | view | reinis
 public | geometry_columns    | view | reinis
 public | raster_columns      | view | reinis
 public | raster_overviews    | view | reinis
 public | spatial_ref_sys     | table | reinis
 public | telpas_eka             | table | reinis
 public | telpas_eka_id_seq      | sequence | reinis
 public | telpas_telpa           | table | reinis
 public | telpas_telpa_id_seq    | sequence | reinis
(26 rows)

rtu_postgis=#
```

Zemāk redzamajā attēlā ir parādītas attieksmes geometry_columns skatā:

Table Relationships

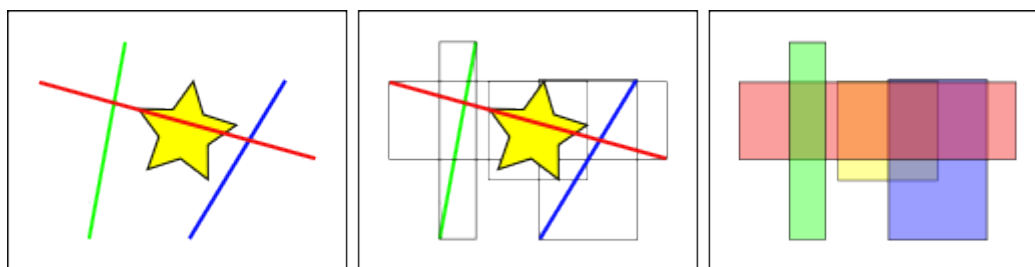


Attēls 4: *geometry_columns* skata struktūra

Indeksi

Parastas relāciju (-objektu) datubāzes indeksāciju veic veidojot hierarhisku koka datu struktūru, balstoties uz indeksējamās kolonnas vērtībām. Telpisko datubāzu indeksi ir savādāki- kompleksas ģeometrijas vietā, tie indeksu veido taisnstūrim, kas iekļauj visu indeksējamo ģeometriju (t.s. *bounding-box*).

Tas redzams sekojošā attēlā:



Attēls 5: Ģeometriskām formām atbilstošie "iekļaujošie taisnstūri"

Pārbaudīt, vai divi taisnstūri pārklājas vai iekļauj viens otru ir algoritmiski ātri- visnotaļ ātrāk, nekā pārbaudīt, vai zilais nogrieznis nekrusto zvaigzni. Kā zināms, viens no labākajiem veidiem kā optimizēt programmas ātrdarbību ir neveikt liekas operācijas. Tāpēc, izmantojot telpiskos indeksus, ir iespējams ātri noteikt, kuras no visām tabulas rindām ir iespējamie kandidāti, un pēcāk- uzmanīgi (un lēni) salīdzināt pirmajā piegājienu izgūtās rindas.

No telpiskajiem indeksiem, PostGIS atbalsta tikai uz R-kociem balstītus indeksus.

Jāatgādina, ka PostgreSQL pēc noklusējuma datus indeksē nevis pēc katras INSERT vai UPDATE operācijas, bet “ik pēc saprātīga laika intervāla”. Tas nozīmē, ka pēc daudzu objektu ievietošanas, ir noderīgi izpildīt sekojošu operāciju, lai tiktu atjaunoti indeksi (un atbrīvota lieki aizņemtā vieta):

```
rtu_postgis=# VACUUM ANALYZE;  
VACUUM
```

Django un GeoDjango

Lai saprastu, kas tieši ir GeoDjango ietvars, vispirms ir jāapjauš, kas ir Django ietvars. Django ir tīmekļa programmatūras ietvars, kas ļauj ātri un ērti izstrādāt tīmekļa programmatūru, tai skaitā, tīmekļa vietnes. Tas ir rakstīts Python programmēšanas valodā, un ļoti lielu uzsvāri liek uz programmatūras koda atkārtotu izmantošanu. Tas ļauj sistēmas komponentes veidot modulāras, vāji saistītas savā starpā, tādējādi atvieglojot to piemērošanu citos projektos (pretstatā stingri saistītām komponentēm, kuras ir grūti atdalīt, lai citā projektā izmantotu tikai vienu komponenti).

Django ietvaram ir iebūvēta funkcionalitāte, kas nepieciešama gandrīz visiem tīmekļa programmatūras projektiem- lietotāju autentifikācija un autorizācija, objektu-relāciju kartēšanas slānis (kas ļoti atvieglo augstāk minēto, modulāro lietojumu realizēšanu), lietotāju sesiju nodrošināšana, pus-automātisks administrācijas interfeiss lietotāja datu modeļiem u.c.

GeoDjango

GeoDjango ir Django paplašinājums, kas paredzēts tieši ģeogrāfisku datu lietošanai tīmekļa vietnēs (kartēšanas lietojumi, lietotāja interfeiss datu analīzes programmatūrai u.c.). Kopš Django 1.5 versijas, GeoDjango projekts ir iekļauts Django instalācijas pakotnē.

GeoDjango sniedz sekojošas iespējas (balstoties uz Django):

- Modeļu lauki ģeometrijas un ģeogrāfijas datu glabāšanai,
- Django ORM slāņa paplašināšana ar atbalstu *spatial* datu manipulācijai un vaicājumu veikšanai,
- Paplašināms administrācijas interfeiss ģeometrijas datiem;

Django instalācija

Tā kā kopš Django 1.5 versijas GeoDjango ir iekļauts Django instalācijas pakotnē, ir nepieciešams uzstādīt Django, kā arī dažas papildu bibliotēkas, kas ļauj no Python koda piekļūt un manipulēt ar ģeometrijas datiem PostgreSQL datubāzē.

Sekojošais piemērs tika izpildīts uz Arch Linux operētājsistēmas ar jaunākajām pakotņu versijām uz šo brīdi.

Vispirms, ir nepieciešamas sekojošas sistēmas pakotnes:

```
$ sudo pacman -S postgresql postgis base-devel python2 python2-virtualenv
```

Šī komanda veiks visu nepieciešamo operētājsistēmas pakotņu instalāciju. Base-devel pakotne satur rīkus, kas nepieciešami Python paplašinājuma kompilēšanai (un lietojot šo operētājsistēmu, šai pakotnei jau būtu jābūt uzstādītai).

Demonstrācijas lietojuma izveide

Virtualenv rīks nav obligāts- to darba autors izmanto, lai vienas operētājsistēmas ietvaros varētu izmantot vairākas Python bibliotēku versijas (lai katram projektam varētu izmantot savu ārējās bibliotēkas X versiju, ja tas nepieciešams).

Izveidosim direktoriju projekta saturam:

```
$ mkdir postgresql
$ cd postgresql/
```

Izveidosim tā saukto virtuālo Python vidi (programmatūrai tā izskatās kā atsevišķa Python instalācija):

```
$ virtualenv-2.7 .env
```

Aktivizēsim šo virtuālo vidi:

```
$ source .env/bin/activate
```

Pārbaudīsim, vai izsaucot komandu python tiks izsaukts python interpretators no virtuālās vides:

```
(.env) $ which python
/home/reinis/RTU/postgresql/.env/bin/python
```

Kā redzams, turpmāk izmantojamais python interpretators atrodas .env/bin direktorijā.

Nepieciešamo Python pakotņu instalācija:

```
(.env) $ pip install django psycopg2
Downloading/unpacking django
  Real name of requirement django is Django
  Downloading Django-1.6.1.tar.gz (6.6MB): 6.6MB downloaded
  Running setup.py egg_info for package django
Downloading/unpacking psycopg2
  Downloading psycopg2-2.5.1.tar.gz (684kB): 684kB downloaded
  Running setup.py egg_info for package psycopg2

Installing collected packages: django, psycopg2
  Running setup.py install for django
    changing mode of build/scripts-2.7/django-admin.py from 644 to 755
    changing mode of /home/reinis/RTU/postgresql/.env/bin/django-admin.py to
755
  Running setup.py install for psycopg2
    building 'psycopg2._psycopg' extension
    gcc -pthread -fno-strict-aliasing -march=x86-64 -mtune=generic -O2 -pipe
-fstack-protector --param=ssp-buffer-size=4 -DDEBUG -march=x86-64
-mtune=generic -O2 -pipe
....
Successfully installed django psycopg2
Cleaning up...
(.env) $
```

Tālāk, līdzīgi kā nodaļā PostGIS, izveidosim jaunu PostgreSQL datubāzi un iespējosim PostGIS paplašinājumu:

```
(.env) $ createdb rtu_postgis
(.env) $ psql rtu_postgis
psql (9.3.2)
Type "help" for help.

rtu_postgis=# CREATE EXTENSION postgis;
CREATE EXTENSION
```

```
rtu_postgis=#
```

Nākošais solis ir Django projekta un Django lietojuma izveide. Atšķirības starp Django projektu un lietojumu ir sekojošas:

Projekts sastāv no viena vai vairākiem lietojumiem. Projektam ir viens konfigurācijas fails, kurā var pievienot jaunus lietojumus, konfigurēt lietojumus, konfigurēt datubāzes.

Savukārt, katrs lietojums ir modulārs- tas var tikt pielietots vienā vai vairākos projektos.

Autora pieredzē ar Django, katram projektam ir viens galvenais lietojums (biznesa funkcionalitātes kodols, iemesls, kāpēc netiek izmantots kāds gatavs risinājums), un vairāki trešās puses lietojumi, kuri tiek pielāgoti konkrēta projekta vajadzībām. Šo trešās puses lietojumu spektrs ir visai plašs- sākot no lietojumiem, kas atvieglo datubāzu migrāciju veikšanu, beidzot ar lietojumiem, kas atvieglo maksājumu pieņemšanu, lietotāju reģistrāciju, kļūdu žurnālēšanu u.c.

Projekta un lietojuma izveidošana:

```
(.env) $ django-admin.py startproject rtu  
(.env) $ django-admin.py startapp telpas
```

Pēc šo komandu izpildes, tiek iegūta sekojošā direktoriju struktūra:

```
(.env)+ postgresql git:(master) x tree  
+  
├── geodjango.odt  
├── postgresq.odt  
└── rtu  
    ├── manage.py  
    └── rtu  
        ├── __init__.py  
        ├── __init__.pyc  
        ├── settings.py  
        ├── settings.pyc  
        ├── urls.py  
        ├── urls.pyc  
        ├── wsgi.py  
        └── wsgi.pyc  
    └── telpas  
        ├── admin.py  
        ├── __init__.py  
        ├── models.py  
        ├── tests.py  
        └── views.py  
├── telpas.jpg  
└── telpas.odg  
  
3 directories, 18 files  
(.env)+ postgresql git:(master) x
```

Nākošais solis ir konfigurēt projektu, lai tiktu izmantota iepriekš izveidotā datubāze un iespējots lietojums 'telpas'.

Failā rtu/settings.py nepieciešamas sekojošās izmaiņas:

```
...  
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',
```

```

'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'django.contrib.gis',
'telpas',
)
..
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'rtu_postgis',
    }
}
...

```

Tālāk, definēsim datu modeļus (tabulas) lietojumā telpas: telpas/models.py

```

from django.contrib.gis.db import models

class Eka(models.Model):
    iela = models.CharField(255)
    pilseta = models.CharField(255)
    lat = models.FloatField()
    lon = models.FloatField()

    kontura = models.MultiPolygonField()
    objects = models.GeoManager()

    def __unicode__(self):
        return "Eka: %d" % (self.pk, )

class Telpa(models.Model):
    eka = models.ForeignKey(Eka)
    stavs = models.IntegerField()
    numurs = models.IntegerField()
    kontura = models.MultiPolygonField()

    objects = models.MultiPolygonField()

    def __unicode__(self):
        return "Telpa: %d" % (self.pk, )

```

Izmantojot manage.py skriptu, ir iespējams apskatīt ģenerētos SQL vaicājumus, kas izveidos nepieciešamo tabulu struktūru.

```

(.env) $ ./manage.py sqlall telpas
BEGIN;
CREATE TABLE "telpas_eka" (
    "id" serial NOT NULL PRIMARY KEY,
    "iela" varchar(255) NOT NULL,
    "pilseta" varchar(255) NOT NULL,
    "lat" double precision NOT NULL,
    "lon" double precision NOT NULL,
    "kontura" geometry(MULTIPOLYGON,4326) NOT NULL

```

```

)
;
CREATE TABLE "telpas_telpa" (
    "id" serial NOT NULL PRIMARY KEY,
    "eka_id" integer NOT NULL REFERENCES "telpas_eka" ("id") DEFERRABLE
INITIALLY DEFERRED,
    "stavs" integer NOT NULL,
    "numurs" integer NOT NULL,
    "kontura" geometry(MULTIPOLYGON,4326) NOT NULL
)
;
CREATE INDEX "telpas_eka_kontura_id" ON "telpas_eka" USING GIST ( "kontura" );
CREATE INDEX "telpas_telpa_eka_id" ON "telpas_telpa" ("eka_id");
CREATE INDEX "telpas_telpa_kontura_id" ON "telpas_telpa" USING GIST
( "kontura" );

COMMIT;

```

Ģenerētie SQL vaicājumi izskatās pareizi. Izpildīsim tos pret datubāzi:

```

(.env) $ ./manage.py syncdb
Creating tables ...
Creating table django_admin_log
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table telpas_eka
Creating table telpas_telpa

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'reinis'):
Email address: reinis@wot.lv
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)

```

Pirmo reizi palaižot datubāzes sinhronizācijas komandu, tiek piedāvāts izveidot administratora lietotāju- šī parole vēlāk būs jāievada administrācijas interfeisā.

Izveidosim administrācijas interfeisu šīm tabulām (fails telpas/admin.py):

```

from django.contrib.gis import admin
from telpas.models import Eka, Telpa

admin.site.register(Eka, admin.GeoModelAdmin)

```

```
admin.site.register(Telpa, admin.GeoModelAdmin)
```

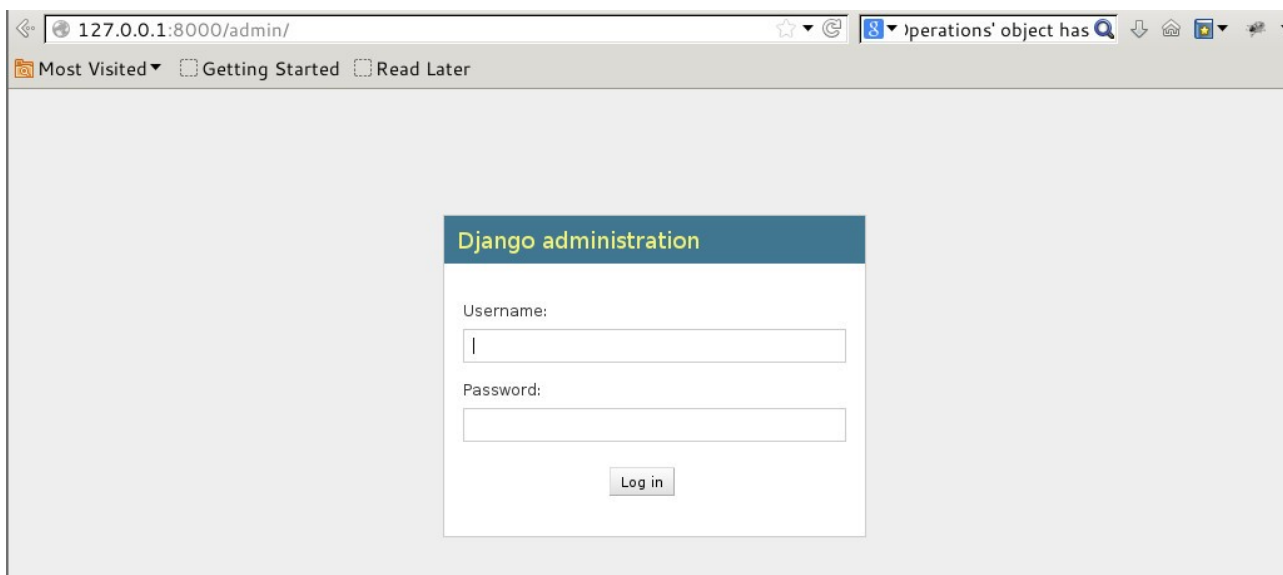
Un visbeidzot, ir jāpalaiž izstrādes serveris:

```
(.env) $ ./manage.py runserver
Validating models...

0 errors found
December 17, 2013 - 11:14:50
Django version 1.6.1, using settings 'rtu.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

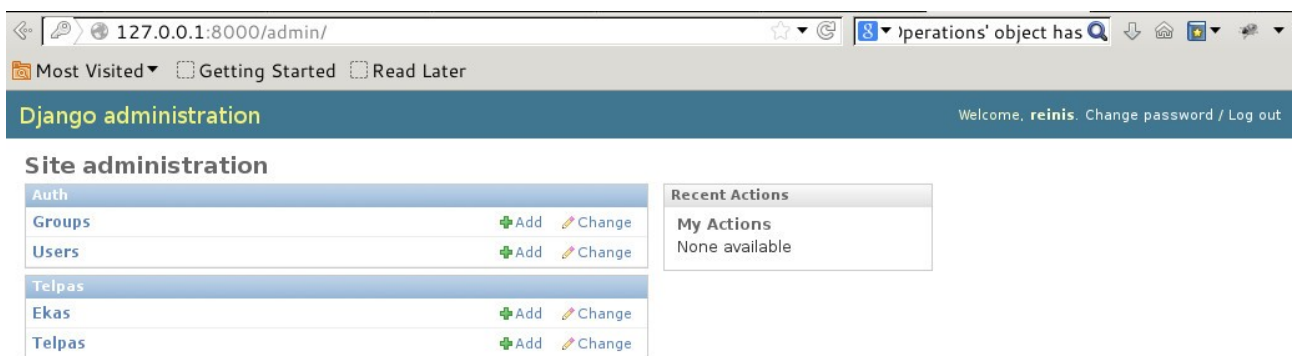
Kad ir palaists izstrādes serveris, jāatver sekojošā adrese tīmekļa pārlūkā:
<http://127.0.0.1:8000/admin>, lai piekļūtu administrācijas interfeisam.

Pēc noklusējuma, tas izskatās sekojoši:



Attēls 6: Administrācijas interfeiss pēc noklusējuma

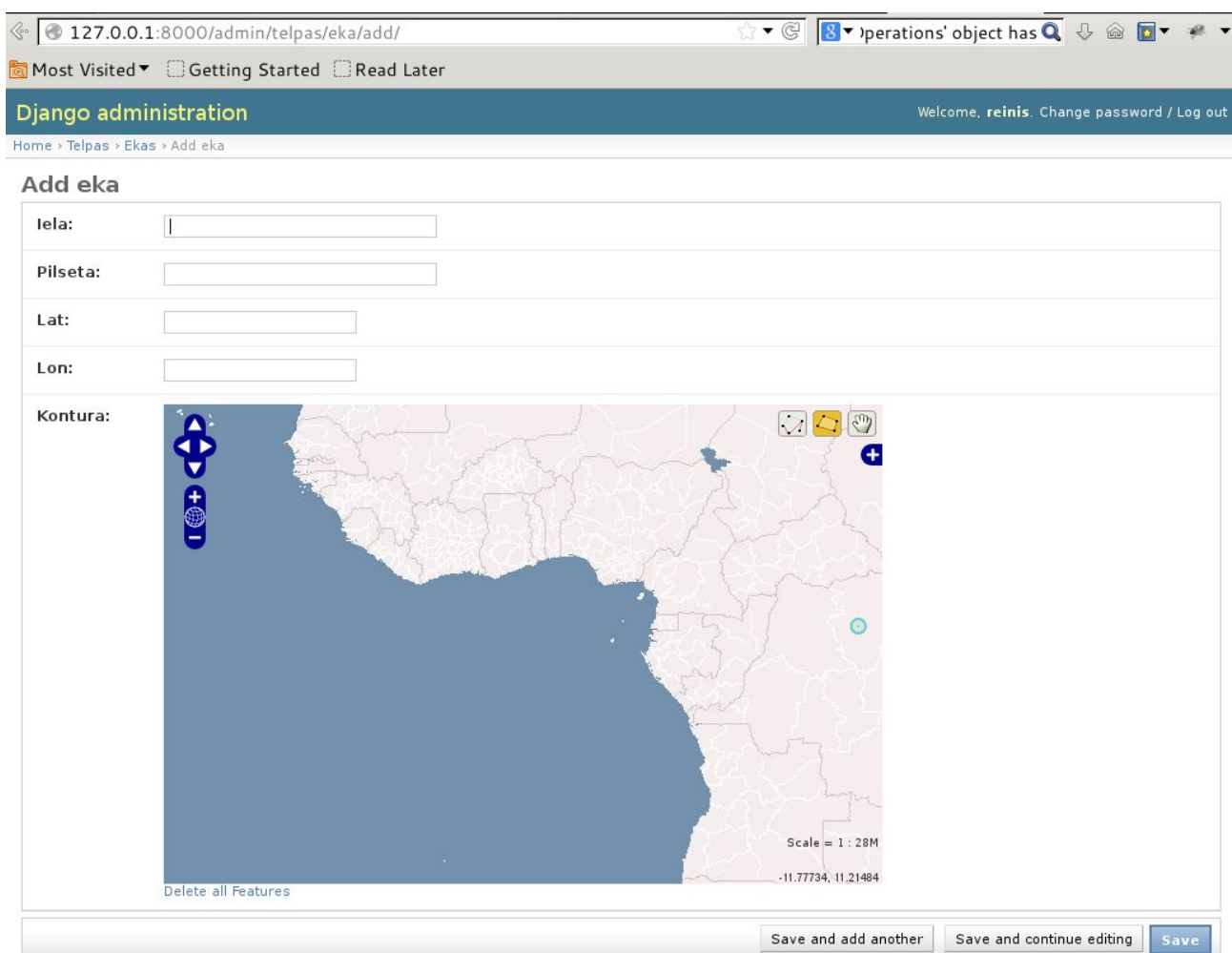
Ievadot datubāzes sinhronizācijas laikā izveidotā lietotāja vārdu un paroli, nokļūstam nākošajā logā:



Attēls 7: Administrācijas interfeisa galvenā lapa

Šajā lapā pēc noklusējuma ir redzami visi administrācijas interfeisam reģistrētie modeļi.

Pamēģināsim izveidot ēku:



Attēls 8: Ēkas pievienošanas logs

Kā redzams, ģeogrāfijas datu ievadīšanai jau tiek piedāvāts ērts komponents- pasaules karte, ar iespējām uz kartes definēt multipoligonu.

Aizpildīsim šo formu un saglabāsim:

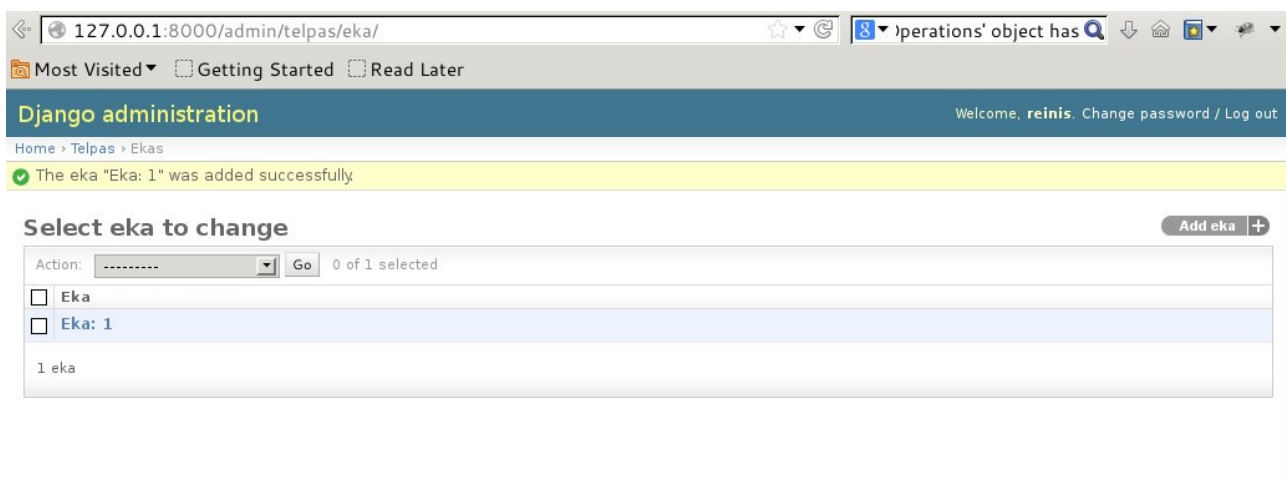
The screenshot shows the Django administration interface for adding a new 'eka' (lake). The form includes the following fields:

- Iela:** Čiekurkalna 1. līnija 84
- Pilsēta:** Rīga
- Lat:** 24.18880
- Lon:** 56.98472
- Kontūra:** A map showing a yellow polygon representing the lake's shape. The map includes navigation controls and a scale of 1:14K.

At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "Save".

Attēls 9: Aizpildīta ēku pievienošanas forma

Kartē izveidotais poligons aprakstīs ēkas ārējo kontūru. Saglabājot šo objektu, dati tiek saglabāti datubāzē:



Attēls 10: Ēku saraksts administrācijas interfeisā

Pārlicināsimies par to, ka ēka patiešām tiek saglabāta datubāzē:

```
(.env) $ psql rtu_postgis
rtu_postgis=# \dt
               List of relations
 Schema |          Name          | Type  | Owner
-----+-----+-----+-----
 public | auth_group             | table | reinis
 public | auth_group_permissions | table | reinis
 public | auth_permission        | table | reinis
 public | auth_user              | table | reinis
 public | auth_user_groups       | table | reinis
 public | auth_user_user_permissions | table | reinis
 public | django_admin_log       | table | reinis
 public | django_content_type    | table | reinis
 public | django_session         | table | reinis
 public | spatial_ref_sys        | table | reinis
 public | telpas_eka             | table | reinis
 public | telpas_telpa          | table | reinis
(12 rows)

rtu_postgis=# \d+ telpas_eka
                                Table "public.telpas_eka"
 Column |          Type          | Storage | Stats target | Description | Modifiers
-----+-----+-----+-----+-----+-----
 id      | integer                |         |               |              | not null default
nextval('telpas_eka_id_seq'::regclass) | plain                  |         |               |              |
 iela    | character varying(255) |         |               |              | not null
 | extended |
 pilseta | character varying(255) |         |               |              | not null
 | extended |
 lat     | double precision       |         |               |              | not null
 | plain   |
 lon     | double precision       |         |               |              | not null
 | plain   |
 kontura | geometry(MultiPolygon,4326) | not null
 | main    |
Indexes:
```

```

    "telpas_eka_pkey" PRIMARY KEY, btree (id)
    "telpas_eka_kontura_id" gist (kontura)
Referenced by:
    TABLE "telpas_telpa" CONSTRAINT "telpas_telpa_eka_id_fkey" FOREIGN KEY (eka_id)
REFERENCES telpas_eka(id) DEFERRABLE INITIALLY DEFERRED
Has OIDs: no

rtu_postgis=# SELECT iela, pilseta, lat, lon, ST_AsText(kontura) FROM telpas_eka;
          iela          | pilseta | lat   | lon   |          st_astext
-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----
Čiekurkalna 1. līnija 84 | Rīga    | 24.1888 | 56.98472 |
MULTIPOLYGON(((24.195585250855 56.986869871618,24.199705123902
56.988414824011,24.201035499574 56.986226141455,24.19704437256
56.984423696997,24.19644355774 56.985582411291,24.198331832887
56.986354887487,24.197902679444 56.987170279028,24.195842742921
56.986354887487,24.1956281662 56.986869871618,24.195585250855 56.986869871618)))
(1 row)
rtu_postgis=#

```

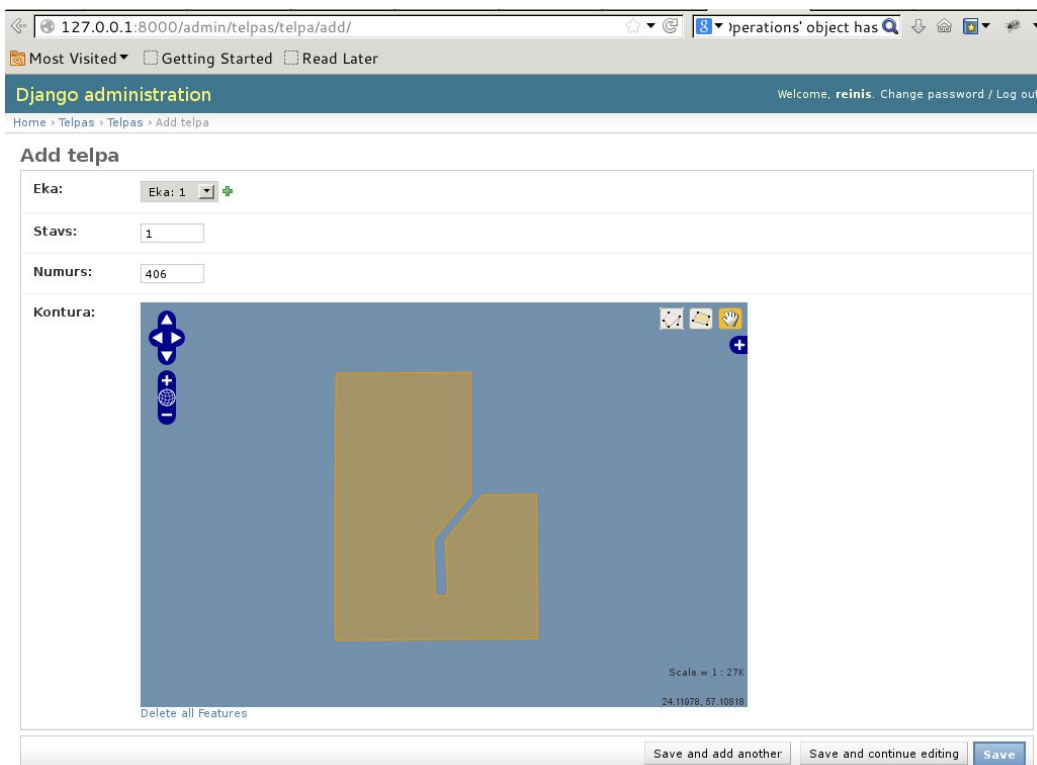
Kā redzams, dati ir tikuši ievietoti datubāzē. Pārskatot tabulas datubāzē, var saskatīt arī tabulas, ko Django ietvars izmanto pēc noklusējuma, lietotāju autorizācijai, autentifikācijai, kā arī lietotāju sesijām.

Turpinot izstrādāt demonstrācijas sistēmu, pārbaudīsim, kā izskatās datu ievades forma telpu plāniem:

The screenshot shows the Django administration interface for adding a new 'telpa' (room) entry. The form includes fields for 'Eka' (set to 1), 'Stavs' (empty), and 'Numurs' (empty). The 'Kontura' field is a map interface showing a blue area with a scale of 1:27K. The interface is titled 'Add telpa' and includes navigation links like 'Home', 'Telpas', and 'Add telpa'. At the bottom, there are buttons for 'Save and add another', 'Save and continue editing', and 'Save'.

Attēls 11: Telpu aprakstošo datu ievades forma

Nodefinēsim sekojošu telpu:



Attēls 12: Datu ievades forma ar izveidotu poligonu

Objektu relāciju attēlojuma slānis

Viens no Django stūrakmeņiem ir objektu-relāciju attēlojuma iespējas (t.s. *object-relational mapping*, turpmāk tekstā, ORM). Tieši šis ORM slānis ļauj pielāgot un integrēt citu izstrādātāju lietojumus Django projektos.

Django projektā, viena no galvenajām vienībām ir modelis. Modelis ir klase, kurai parasti ir atbilstoša datubāzes tabula, un modeļa klases lauki parasti atbilst tabulas kolonnām. Izmantojot Python programmēšanas valodas iespējas, ir iespējama tradicionāla objektorientētas programmēšanas pieeja datu struktūru modelēšanai - modelis var mantot laukus un metodes no cita modeļa, vai nu izveidojot jaunu tabulu (ja nepieciešami papildu lauki), vai arī izmantojot vecāka modelim atbilstošo tabulu (ja mantošanu izmanto, lai paplašinātu klases metodes). Django ORM slānis atbalsta arī abstraktus bāzes modeļus (abstrakta klase ir klase, no kuras var mantot citas klases, bet nevar izveidot abstraktās klases objektus).

No šīm modeļu klasēm, Django var automātiski ģenerēt un izveidot datubāzes tabulas projekta uzstādījumos noteiktajai DBVS. Izmantojot trešās puses lietojumus, var ģenerēt arī migrāciju skriptus modeļa struktūrai un, ja nepieciešams, arī datiem. Datu struktūras migrāciju skriptu izveidošana lielākoties ir automatizējams process, ja izmaiņas ir nelielas. Kardinālas datu struktūras maiņas gadījumā, šos migrāciju skriptus ir iespējams rakstīt pašam.

Izmantojot iepriekšējos piemēros definēto modeļa klasi, tiek demonstrētas sekojošas objektu-relāciju attēlojuma iespējas Python interaktīvajā konsolē:

```
$ ./manage.py shell
Python 2.7.6 (default, Nov 26 2013, 12:52:49)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from telpas.models import Eka
>>> Eka.objects.all()
[<Eka: Eka: 1>]
```

Modeļu **objects** atribūts ir ieteicamais veids kā piekļūt pie datubāzes, un izpildīt vaicājumus, kuri atgriež modeļu objektus. Metode **objects.all()** atgriež visus objektus datubāzē. Atgrieztā vērtība ir t.s. *lazy object*- tas nozīmē, ka vēršanās pie datubāzes notiks tikai mirklī, kad objekti tiks izmantoti. Tā, piemēram, var izsaukamām procedūrām nodot vaicājumu, kuru procedūra, iekšienē, atkarībā no vajadzības, vēl var labot. Piemēram, ja procedūrai interesē tikai atgriezto objektu skaits, no datubāzes netiks ielasītas visas rindas, bet vaicājums automātiski tiks pārveidots par *SELECT COUNT(*) FROM..* vaicājumu, kas parasti izpildās ātrāk.

```
>>> e = Eka.objects.first()
>>> e
<Eka: Eka: 1>
>>> dir(e)
[... clean', 'clean_fields', 'date_error_message', 'delete', 'full_clean', 'id', 'iela', 'kontura', 'lat', 'lon', 'objects', 'pilseta',
'pk', 'prepare_database_save', 'save', 'save_base', 'serializable_value', 'telpa_set', 'unique_error_message',
'validate_unique']
>>> e.iela
u'\u010ciekurkalna 1. l\u012bnija 84'
>>> e.kontura
<MultiPolygon object at 0x1bc90a0>
>>> e.lat
24.1888
>>> e.pilseta
u'R\u012bga'
```

Mainīgajā ar nosaukumu **e** ievietojam pirmo datubāzes atgriezto objektu. Iebūvētā Python funkcija **dir()** ļauj apskatīt objekta īpašības un metodes. Redzam, ka piekļuve pie modeļa laukiem ir ļoti vienkārša. Daudzpunktes vietā ir redzamas arī objekta privātās metodes, bet tās ir izlaistas rezultātu uztveramības labad.

Redzam arī, ka kolonna **kontura** glabā **MultiPolygon** klases objektu. Apskatīsim šī objekta īpašības:

```
>>> dir(e.kontura)
[... 'append', 'area', 'boundary', 'buffer', 'cascaded_union', 'centroid', 'clone', 'contains', 'convex_hull', 'coord_seq',
'coords', 'count', 'crosses', 'crs', 'difference', 'dims', 'disjoint', 'distance', 'empty', 'envelope', 'equals', 'equals_exact',
'ewkb', 'ewkt', 'extend', 'extent', 'geojson', 'geom_type', 'geom_typeid', 'get_srid', 'has_cs', 'hasz', 'hex', 'hexewkb',
'index', 'insert', 'interpolate', 'interpolate_normalized', 'intersection', 'intersects', 'json', 'kml', 'length', 'normalize',
'num_coords', 'num_geom', 'num_points', 'ogr', 'overlaps', 'point_on_surface', 'pop', 'prepared', 'project',
'project_normalized', 'ptr', 'ptr_type', 'relate', 'relate_pattern', 'remove', 'reverse', 'ring', 'set_srid', 'simple', 'simplify',
'sort', 'srid', 'srs', 'sym_difference', 'touches', 'transform', 'tuple', 'union', 'valid', 'valid_reason', 'within', 'wkb', 'wkt']
```

Pie datiem, kas tiek glabāti kolonnā var tikt izmantojot iebūvēto Python funkciju, **print()**.

```
>>> print(e.kontura)
MULTIPOLYGON (((24.1955852508550002 56.9868698716179978, 24.1997051239019996 56.9884148240110022,
24.2010354995740009 56.9862261414549991, 24.1970443725600006 56.9844236969969984,
24.1964435577400003 56.9855824112909986, 24.1983318328869998 56.9863548874870034,
24.1979026794439989 56.9871702790279997, 24.1958427429209983 56.9863548874870034,
24.1956281661999988 56.9868698716179978, 24.1955852508550002 56.9868698716179978)))
```

Pamēģināsim izmantot nedaudz sarežģītāku vaicājumu. Uzdevums- atrast ēkas, kas atrodas X km rādiusā. Lai to izdarītu, vispirms definēsim atskaites punktu:

```
>>> from django.contrib.gis.geos import *
>>> point = fromstr('POINT(24.879684 57.146671)', srid=4326)
```

Punkts ar šīm koordinātēm atrodas Siguldas novadā. Pamēģināsim atrast ēkas, kas atrodas 10 km rādiusā ap šo punktu:

```
>>> Eka.objects.filter(kontura__distance_lte = (point, 10000))
[]
```

Metode **objects.filter()** ļauj meklēt datubāzē objektus, kas atbilst kādiem nosacījumiem. Šīs metodes argumentu semantika ir **lauks = vērtība**. Tā, piemēram, šādi varētu atrast objektus, kuru lauks **pilseta** ir vienāds ar “Rīga”:

```
>>> Eka.objects.filter(pilseta=u"Rīga")
[<Eka: Eka: 1>]
```

Iepriekšējā piemērā, **kontura__distance_lte = (point, 10000)** nozīmē- atlasīt objektus, kuru lauks **kontura** ir 10000 metru rādiusā no punkta **point**. Tā kā Sigulda no Rīgas atrodas aptuveni 50km attālumā, ir loģiski, ka rezultātu nav. Paplašināsim meklēšanas rādiusu līdz 60km:

```
>>> Eka.objects.filter(kontura__distance_lte = (point, 60000))
[<Eka: Eka: 1>]
```

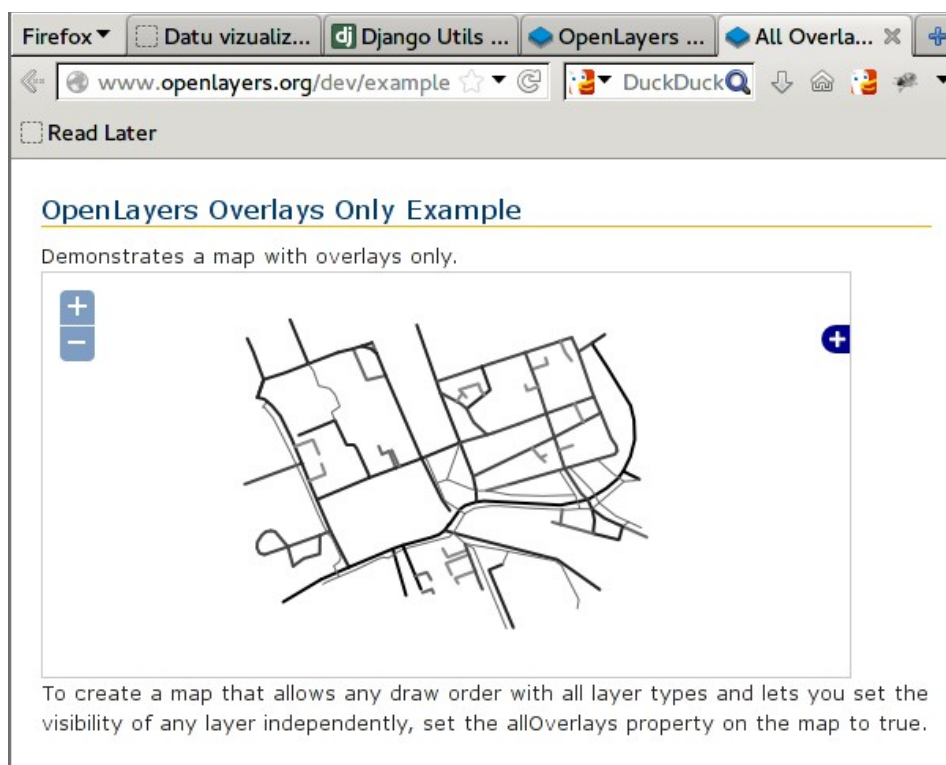
Tagad vaicājums veiksmīgi atgriež objektu, kura poligons atrodas Rīgā.

Datu vizualizācija ar GeoDjango

GeoDjango nenodrošina datu vizualizāciju, atskaitot administrācijas interfeisu. Lai vizualizētu ģeotelpiskos datus, jāizmanto trešās puses komponentes, piemēram, OpenLayers.

OpenLayers

OpenLayers ir komponente, kas paredzēta kartogrāfiskas informācijas attēlošanai tīmekļa pārlūkā. Šī komponente tiek izmantota OpenStreetMap projektā, kurā lietotāji paši var uzturēt brīvpieejas karšu sistēmu. Tajā, līdzīgi kā Wikipedia, lietotāji paši var rediģēt kartes (mainīt ielu nosaukumus, veidot no jauna izbūvētas ielas, interešu punktus u.c.).



Attēls 13: OpenLayers interfeiss bez pasaules kartes fonā

Datu vizualizācija ar trešās puses komponenti iesākumā var šķist sarežģīta, bet patiesībā ir jāraksta tikai kods, kas savieno šo komponenti ar GeoDjango. Eksistē arī gatavi Django papildinājumi kartogrāfiskās informācijas attēlošanai ar GeoDjango, bet tie šajā darbā netiek apskatīti, lai izprastu vizualizācijas būtību.

OpenLayers atbalsta vairākus formātus priekš kartogrāfiskās informācijas attēlošanas: KML, GML, GeoJSON, GeoRSS un citus. Katram no šiem formātiem ir savas priekšrocības un trūkumi, kuri diemžēl sīkāk izskatīti netiek. Šī darba nolūkiem tiek izmantots GeoJSON formāts, jo to ir viegli pielietot tīmekļa lietojumos (JSON, jeb JavaScript Object Notation ir *de facto* datu apmaiņas standarts tīmekļa lietojumos JavaScript valodā, savukārt GeoJSON ir JSON apakškopa).

Turpmāk esošais piemērs ir atrodams šim dokumentam pievienotajā Django projektā, failā `telpas/views.py`.

Sāksim ar nepieciešamo datu izgūšanu no datubāzes:

```
qs = Eka.objects.all()
```

Tālāk, izmantosim `vectorformats` bibliotēku, lai nodefinētu, no kāda formāta (un kurus atribūtus) vēlamies pārveidot:

```
from vectorformats.Formats import Django, GeoJSON
djf = Django.Django(geodjango="kontura", properties=['lat', 'lon'])
```

Pēc tam, izmantojot `vectorformats` bibliotēku, nodefinēsim formātu, uz kuru vēlamies pārveidot (GeoJSON).


```
geoj = GeoJSON.GeoJSON()
```

Un visbeidzot, atliek izsaukt metodes, kas vispirms dekodē vaicājuma rezultātus uz kādu vectorformats iekšēju datu attēlojumu, un pēc tam- enkodē GeoJSON formātā.

```
s = geoj.encode(djf.decode(qs))
```

Pēcāk, atliek tikai šos datus ievietot šablonā “telpas/index.html”, un rezultējošo HTML kodu atgriezt tīmekļa pārlūkam.

```
from django.utils.safestring import mark_safe
data = {'geojson': mark_safe(s)}
return render(request, 'telpas/index.html', data)
```

Šablons, arī ir diezgan vienkāršs- apskatīsim svarīgākās šablona daļas (atrodas failā telpas/templates/telpas/index.html):

```
10 <link rel="stylesheet" href="{% static 'css/openlayers/default/style.css"
%}" type="text/css">
11 <script src="{% static 'js/0penLayers.js" %}"></script>
```

Iekļaujam OpenLayers JavaScript kodu, kā arī OpenLayers stila izkārtojumu.

```
29 <script type="text/javascript">
30     var lon = 5;
31     var lat = 40;
32     var zoom = 5;
33     var map, layer;
34
35     function init(){
36         OpenLayers.ImgPath = "/static/css/openlayers/default/img/";
37
38         map = new OpenLayers.Map('map');
39         layer = new OpenLayers.Layer.WMS( "OpenLayers WMS",
40             "http://vmap0.tiles.osgeo.org/wms/vmap0",
41             {layers: 'basic'} );
42         map.addLayer(layer);
43         map.setCenter(new OpenLayers.LonLat(lon, lat), zoom);
44         var featurecollection = {{geojson}};
45         var geojson_format = new OpenLayers.Format.GeoJSON();
46         var vector_layer = new OpenLayers.Layer.Vector();
47         map.addLayer(vector_layer);
48
49         vector_layer.addFeatures(geojson_format.read(featurecollection));
50     }
51 </script>
```

Šajā koda fragmentā ir redzama OpenLayers inicializācija. Treknrakstā ir izcelta šablona daļa, kas tiks aizstāta ar datiem, ko vēlamies attēlot uz kartes.

Izdarot iepriekš minētās darbības, var nonākt pie šāda rezultāta:



Secinājumi

PostgreSQL ir viena no iespējam bagātākajām atvērtā pirmkoda relāciju datubāzu vadības sistēmām. Autora pieredze projektu izstrādei izmantojot PostgreSQL kā datubāzu vadības sistēmu ir bijusi visnotaļ pozitīva, it īpaši, ņemot vērā alternatīvas (MySQL, SQLite). PostgreSQL pēc noklusējuma ir strikta konfigurācija uzstādījumos, kas attiecas uz datu integritāti.

PostgreSQL ir iebūvēti datu tipi vienkāršu ģeometrisku datu glabāšanai, taču iespējas ar šiem datiem manipulēt ir diezgan trūcīgas. Kā viens no trūkumiem ir jāmin arī sfēriskās koordinātu sistēmas trūkums.

PostGIS ir paplašinājums PostgreSQL, kas nodrošina gan ģeometrijas (Dekarta koordinātu sistēmā) gan ģeogrāfijas (sfēriskajā, WGS 84 koordinātu sistēmā) datu glabāšanu un apstrādi.

PostGIS jau varētu tikt izmantota produkcijas vidēs- izstrādājot šo darbu autoram jau radās pielietojums, kurā pašreiz tiek izmantota relāciju datubāze, taču telpiskā datu bāze tam ir piemērotāka, un PostGIS piedāvātās iespējas šķiet pietiekamas problēmas risināšanai.

Darba praktiskā daļa tika izstrādāta tīmekļa programmatūras izstrādes ietvarā Django. Šī ietvara iespējas ir piemērotas ātrai projektu izstrādei, pateicoties gan iebūvētajai funkcionalitātei, kas nepieciešama gandrīz katrai tīmekļa vietnes sistēmai, gan arī pateicoties modulārajiem uzbūves principiem, kas ļauj atkārtoti izmantot dažādus, citur izstrādātus lietojumus.

Datu vizualizācijai tika izmantots trešās puses komponents OpenLayers, jo Django nav iebūvētu mehānismu topogrāfisko datu vizualizācijai. Šāda, atsevišķa komponenta izmantošana varētu šķist apgrūtināša, tomēr jāņem vērā, ka Django projekta autoru filozofija ir neieklāut visu kas var būt noderīgs- tikai to, kas ir nepieciešams teju katram tīmekļa lietojumam.

Šī darba laikā izstrādātais Django projekts ir pieejams tiešsaistē:
<https://github.com/festlv/geodjango-example>

Izmantotā literatūra

<http://postgis.net/docs/manual-2.1/>

<http://www.postgresql.org/docs/9.3/interactive/index.html>

<https://en.wikipedia.org/wiki/PostGIS>

<http://workshops.boundlessgeo.com/postgis-intro/geometries.html>

<https://docs.djangoproject.com/en/dev/ref/contrib/gis/>