Spatializing your Data with PostGIS, GeoDjango, and OpenLayers

Chander Ganesan

O'Reilly OSCON

July 21, 2009

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Table of Contents

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

What's the Big Deal with GIS?

- GIS – Geographic Information Systems
  - Designed to add the "where" to the "what", "when", "how", and "why" of things.
  - Uses industry standard terms and mechanism to allow people to share data dealing with location on a planet (typically, Earth, but works for your planet of choice).
  - Used to answer questions like:
    - How many Republicans are in a voting district?
    - What city is my plane currently flying over?
    - What's the average value of land in San Jose?
    - How can I re-draw school district lines to ensure that my school isn't overcrowded?
    - What's the shortest route from X to Y?
    - Where is my iPhone right now?

GIS Fundamentals

What is GIS?

- GIS or Geographical Information Systems are systems that are used to capture, store, display, and analyze geospatial data, and its associated attributes.
  - The term *geospatial* refers specifically to the planet Earth (or some other planet) as the basis of its coordinate system.
  - GIS tools allow us to generate queries (against a database, in most cases) that allow us to analyze spatial information, edit data, maps, and display or store the results of such operations.
- Most GIS data includes two components:
  - A spatial component – this describes the location or spatial distribution of some kind of geography.
    - Spatial components are typically represented using one of two methods:
      - As a field representation. In a field representation each point in the space has an assigned value – this leads to a *raster data model*. Raster models use a matrix, where each cell contains a value (for example, an elevation or other properties).
      - As a set of points that are strung together to make lines or polygons. This is known as a *vector data model.* PostGIS and GeoDjango are both designed to use vector data.
  - An attribute – which is used to describe the properties of the spatial component.

Vector Data

- GeoDjango and PosGIS both store and manage spatial data using a Vector model.
- Unlike raster data, which is made up of "boxes" or "pixels", vector data is comprised of lines or arcs.
  - Each line or arc is defined by a beginning or end points, with each set of points meeting at a node.
  - The node location itself, and the topological structure usually stored explicitly.
- Features are defined by their boundaries, and curved lines are represented as a series of connecting arcs.
- Geo spatial data in a vector format is stored as a set of coordinates. The basic units used to represent this information is:
  - Points – a zero-dimensional abstraction of an object, represented by a singly X, Y coordinate set.
  - Lines – a set of coordinates that represent the shape of a geographic feature too narrow to be displayed as an area (such as a street, runway, sewer line, or stream).
  - Polygon – a feature that is used to represent an area. A polygon is defined by lines that make up its boundaries. Additionally a polygon will have a point defined inside its boundary to be used for identification.

Spatial Reference Systems

- The spatial reference system (SRS) is a combination of the Projection and the Datum being used.
  - Basically, it defines an ellipsoid.
  - It then defines a datum using that ellipsoid.
  - Lastly, it defines a geocentric, geographic, or projection.
- The projection being used always has a geographic coordinate system associated with it.
  - The European Petroleum Survey Group (EPSG) has a set of pre-defined spatial references, each of which has been assigned a unique ID (an EPSG number).
  - PostGIS uses a *spatial reference identifier* (SRID) to indicate the spatial reference used by a geometry.
- So whenever we talk about Geospatial data, we are referring to the following as a point of reference:
  1. An Ellipsoid
  2. A Datum using that ellipsoid (the datum provides the center and orientation for the ellipsoid by mapping it on to the surface of the planet at a particular point in time).
  3. Some projection (actually, we could have it defined as geocentric or geographic as well).
- Usually we use a standardized EPSG spatial reference system for the geographic coordinate system.

About PostGIS

- PostGIS is a set of extensions to the PostgreSQL Object-Relational Database system (the world's most advanced Open Source Database).
- PostGIS adds spatial datatypes, operators, and functions to PostgreSQL.
  - An indexing strategy to quickly and easily locate features relative to other features.
  - A set of functions to perform re-projection of data from one spatial reference system to another.
    - Thus allowing table "A" to use one EPSG value and "B" to use another, and "joins" to be performed between geometries in the two.
  - A set of functions to construct a new geometry from existing ones
    - Merge two geometries together.
    - Expand a geometry out to make it bigger.
  - A set of functions to perform geometric tests (what is the distance between these two geometries, does this geometry overlap this other geometry, etc).
- Closest analogous tools would be:
  - ESRI Spatial Database Engine (SDE) ($$$)
  - Oracle Spatial ($$$)
  - MySQL Spatial (immature/feature poor)
- Used by lots of bug guys...
  - Natural Resources Canada, US Navy, US Army, NASA, NOAA – just to name a few
- Can do path routing using tools like PgRouting.

About GeoDjango

- GeoDjango is an extension of the Django Model-Template-View (MTV) web development framework.
  - GeoDjango provides a number of modules, classes, and tools that may be used to quickly develop and deploy web applications that use geospatial technologies.
- Like Django, GeoDjango is designed to be fast and easy, and lets you build high-performing, elegant Web applications quickly
- GeoDjango focuses on a philosophy known as "Don't Repeat Yourself" (DRY), which states that every piece of knowledge must represented in a single, unambiguous, and authoritative way in the system.
  - To this end, GeoDjango eschews the duplication of code.
- GeoDjango expands upon Django's framework by adding a number of models associated with geospatial technologies, along with some related tools to import, export, and manage data.
- GeoDjango's admin interface provides an OpenLayers interface that may be used to modify and manage spatial data stored in the database.
- GeoDjango allows developers to build powerful application using PostGIS without knowing SQL, PostGIS specifics, etc.

GeoDjango Components

- The GeoDjango system consists of several integrated components:
  - GeoDjango, like Django, is written in Python, and requires a Python version of 2.4 or greater.
  - Django - the web framework that provides the basis for GeoDjango, Django provides:
    - The Object-relational mapper (ORM) allows developers to define data models in Python, and use a dynamic database-access API to connect to and manage data in the database.
    - An Automatically generated administrative interface.
    - Elegant URL Support
    - Built-In Templating System
    - Built-in Hooks to Cache systems
    - Support for multi-language applications
  - A spatial database
    - Currently, GeoDjango provides support for PostgreSQL (with PostGIS), MySQL, and Oracle Spatial.
    - You'll likely find the best set of features and support with the PostGIS spatial extensions to PostgreSQL.
  - GeoDjango uses several Geospatial libraries::
    - GEOS - An Open Source Geometry Engine (also required for PostGIS)
      - This may be downloaded and installed from http://trac.osgeo.org/geos/
    - PROJ.4 - A cartographic projection/reprojection library

In this tutorial, we'll focus exclusively on using the PostGIS spatial extensions to PostgreSQL.

- This may be downloaded and installed from http://trac.osgeo.org/proj/

- GDAL - A Geospatial Data Abstraction Library (used for import/export of data, among other things).
  - This may be downloaded and installed from http://www.gdal.org/

Installing & Configuring GeoDjango Components

GeoDjango Pre-Requisites

- Django is based on Python, and as such requires Python as a pre-requisite.
  - As of 11/2007, Django does not provide support for Python3000's upcoming release, and it's unlikely that Python3000 will be supported for some time after release.
  - GeoDjango, at a minimum, requires python 2.4 or higher - though it is recommended you use the latest stable 2.x release of Python when possible.

GeoDjango is now included in the mail branch of Django code. As such, the installation of GeoDjango is the same as the installation of Django.

- You can run `python --version` to determine the version of Python you currently have, and upgrade if necessary.
- Django integrates closely with the web server, and in particular, it is used to process any files that are requested via the server.
  - As such, in order to use Django, you'll need to have a web server installed.
- Django utilizes a database for the data model.
  - You'll need to have a database installed in order to use Django. Supported databases right now include PostgreSQL, MySQL, and SQLite.
  - The recommended (and arguably the most common database used with Django) is PostgreSQL.

Django does have its own web server you can use for testing, but its generally recommended that you use an enterprise-grade server, such as Apache or IIS.

- In order to work with your database, you'll need to ensure that the appropriate database driver is installed. In the case of PostgreSQL, you'll need 'psycopg2' (http://www.initd.org).

- GeoDjango requires that the following components be installed:
  - GEOS - An Open Source Geometry Engine (also required for PostGIS)
    - This may be downloaded and installed from http://trac.osgeo.org/geos/
  - PROJ.4 - A cartographic projection/reprojection library
    - This may be downloaded and installed from http://trac.osgeo.org/proj/
  - GDAL - A Geospatial Data Abstraction Library (used for import/export of data, among other things).
    - This may be downloaded and installed from http://www.gdal.org/

Choosing a Database

- Current version of GeoDjango provide support for three databases:
  - PostgreSQL using the psycopg and psycopg2 drivers. The recommended driver is psycopg2, but you can use version 1 as well. In this tutorial, we'll use PostgreSQL with psycopg2
  - MySQL using the MySQLdb package. This package is available at http://sf.net/projects/mysql-python/ .
  - Oracle using the proprietary Oracle drivers
- GeoDjango supports all of these databases in the same manner, so which you choose to use here makes little difference, since everything is coded in the same way.
- When you move to your own development system, you'll need to choose a database based on your needs and the software deployed in your enterprise.

Downloading & Installing Django

- Django can be downloaded from http://www.djangoproject.com/download/
  - We have the option of downloading the latest official release, which is the recommended route.
  - If you feel like living on the edge, you can also download the "work in progress" release from the Subversion source code control system.
    - In order to download via subversion, you'll need to have a subversion client installed and configured as well.
- Once you've downloaded the latest Django archive, you'll need to unpack and install it.
  - Unpacking it involves using the 'tar' command to extract the source files.
  - Once this is done, you'll need to cd into the Django source directory and issue the command below as the user 'root':

    `python setup.py install`
    - Be sure you are logged in as root (or using the sudo command) when you issue this on the Linux/Unix platform, since Django will require administrative rights for the installation process.
    - This command works the same on Windows as it does in the Linux environment, you just need to make sure you call the correct interpreter.

```
import psycopg2
import django
print django.VERSION
print psycopg2.__version__
```

Sample code to verify that psycopg2 and Django are installed correctly.

Creating a PostGIS Database

- In order to create a PostGIS database, we need to do a few things (in the steps below, we'll create a PostGIS database called 'mygeo'):

1. Connect to your running instance of PostgreSQL (via the psql client) and issue the 'create database' command.
   ```
   Create database mygeo;
   ```
2. From the command line (not psql) prompt, add the PL/PgSQL language to the new database:
   ```
   PGBASE/bin/createlang plpgsql mygeo
   ```
3. Use the psql tool to load the 'lwpostgis.sql' file into your database:
   ```
   PGBASE/bin/psql -d mygeo -f /usr/local/pgsql/share/lwpostgis.sql
   ```
4. Load the EPSG coordinate system information into the database:
   ```
   PGBASE/bin/psql -d mygeo -f /usr/local/pgsql/share/spatial_ref_sys.sql
   ```

- Once you've created the database, and installed the objects, it is possible to use the database as a "template" for future databases.
  ```
  create database usageo with template mygeo;
  ```
  - The command above would create the 'usageo' database by cloning the objects already in 'mygeo'.
  - Note that if you store additional data in mygeo, it would get cloned as well.

While it is possible to clone an entire database and its contents using the "with template" statement, doing so is generally not recommended. Use the command to only clone "basic" databases that are "templates" for others.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Creating GeoDjango Models

Creating a New Project

- When you start a new Django project, there are a few things you should note:
  - First, your project files won't (should not) exist under the document root of your web server.
    - We'll configure the web server later - so it can find what it needs - but generally, putting your code under a web-accessible directly makes it less secure.
  - You'll need to create a directory to store your project, and then use the django-admin.py script to create the new project files.
    `django-admin.py startproject name`
    - The startproject argument tells the command that we need it to initialize a new Django project.
    - The *name* argument allows us to specify the name of the project to be created.
    - When the command runs, it will create a new directory called *name* in the current directory, and populate it with a few files:
      - **`__init__.py`** : A file that causes Python to treat the directory as a package.
      - **`manage.py`**: A command line utility to communicate with Django.
      - **`settings.py`**: Configuration settings for the project.
      - **`urls.py`**: URL declarations for the project. The URL's tell Django what to do when certain URL's are put into the browser.

On Unix (and Linux) systems, the django-admin.py command will be an executable under your system PATH. As a result, you can just type in the command to run it. On Windows systems, this script normally gets installed in the 'Scripts' directory under your Python installation, so you'll need to either add that directory to your search path, or run the command by specifying the full path to the file.

- Once you've created your project, you can test it by starting up the Django development server.
  - You can start the server using the command:
    `python manage.py runserver`
    - This command will start up the server and listen on port 8000 of your system.
    - You can use your web browser to connect to the server using the URL the command returns.
- The development server is ideal for development, since it watches your code for changes, and reloads the files whenever necessary.

```
/> python.exe manage.py runserver
Validating models...
0 errors found

Django version 1.0.2 final, using settings 'demo.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Running the Django development web server.

  - This allows you to make changes and update your code, and test it directly via the browser - without having to restart services, etc.

Adding in GeoDjango

- Since GeoDjango is a contributed application to Django, you'll need to modify your settings.py so that '`django.contrib.gis`' is one of the `INSTALLED_APPS`.
  - Without this, many of the geographic features, KML sitemaps, etc. may not be available.
- In order to use the Geographic Admin interface with OpenStreetMaps (as the base layer), then you must add the Spherical Mercator projection into your PostGIS installation, this is done using the following:

```
./manage shell
from django.contrib.gis.utils.srs import add_postgis_srs
add_postgis_srs(900913)
```

  - Be sure that the user that is accessing the database has write access/permissions to the spatial_ref_sys table, otherwise it will fail to perform the insert correctly.

You'll need to modify your settings.py file as appropriate with credentials to connect to PostgreSQL, template directory locations, and other related data.

Those steps aren't covered in this tutorial, but are all contained in settings.py.

Creating a New Application

- Once you've created a project, you'll need to create an application.
  - Django models and views are handled through the application itself – not the project, which consists of primarily the controller and base settings (such as data store information)
- You can begin a new app by opening a shell window in your project directory and issuing the 'startapp' command.

```
manage.py startapp wake
```

  - Like projects, you should avoid using reserved words for your app name, otherwise you might run into problems later.
- Once you've created a new app (or installed an app from a third party), you'll want to add it to the list of INSTALLED_APPS in your projects 'settings.py' file.
  - Apps are not actually used until they are installed, so you'll want to make sure you install your app, or it won't be usable in your project.

Importing Spatial Data

The GeoDjango Data Model

- In order to support GeoSpatial databases, GeoDjango provides some extensions to the normal Django data model:

Model Field	Description
`PointField`	A field that will contain point data.
`LineStringField`	A field that will contain line string data.
`PolygonField`	A string that will contain a single polygon.
`MultiPointField`	A field that will contain multiple points.
`MultiLineStringField`	A field that will contain multiple line strings.
`MultiPolygonField`	A field that will contain multiple polygons.
`GeometryCollectionField`	A field that will contain a collection of Line Strings, Points, and Polygons.

  - Each of these fields has two **named** arguments, one of which is required:
    - srid – The Spatial Reference System Identifier (SRID) for the field. If not provided, this defaults to 4326 (WGS84).
    - spatial_index – Indicates whether an index should be created for this field (defaults to true)
- Generally, your data will dictate the field type to be used.

The GeoManager

- In order to construct spatial queries, each model that utilizes spatial fields must contain a GeoManager model manager.
- The model manager is required to construct proper spatial queries, and without it, spatial queries will fail.
- **A GeoManager model is required for any model that either utilizes spatial data fields, or has a relationship (such as ForeignKey) to a table that utilizes these fields.**
  - If a model does not contain a GeoManager, and has no spatial fields, it will not be usable in queries that leverage relationships between it and spatial database fields.

```
class fireresponse(models.Model):
    district = models.CharField(max_length=254)
    area = models.FloatField()
    len = models.FloatField()
    objects = models.GeoManager()
    geom = models.MultiPolygonField(srid=2264,spatial_index=True)
    class Meta:
        verbose_name_plural = 'Fire Response Unit Areas'
    def __str__(self):
        return self.district
```

A sample GeoDjango model using a MultiPolygonField with SRID 2264 and a spatial index.

Using ogrinspect to Generate Models & Mappings

- Since we can programmatically determine the fields, properties, and other shapefile data, it stands to reason that we can write code to have Geodjango automatically do this for us.
- The ogrinspect tool automatically (when given some arguments) is able to read an import file, and generate a LayerMapping and model definition from it.
  - Once we have these things, its only a few steps more to actually use the Model and mapping to import data into our database.
- Ogrinspect is an argument that is passed into the manage.py command, and accepts some arguments:
  - A data file to inspect (the shapefile)
  - The name of the model to generate
  - Any arguments/options to pass in.
    - Arguments are preceded by a set of double hyphens, and are typically:

Shapefiles are made up of 4 different component files:
The .shp file holds geometry data.
The .shx file holds a spatial index (and is not required)
The .dbf file contains object meta-data.
The .prj file contains projection information.

The ogrinspect tool generates a model that matches a given shapefile, thus saving us the work of defining our model by hand.

The tool also generates a LayerMapping – a mapping that controls the import of data from the shapefile into the database.

Using ogrinspect, we can skip the tedious process of defining both of these manually.

- --srid=#### - To specify the SRID for the geographic field in the model.
- --mapping – Tells ogrinspect to generate the model and mapping (otherwise it just does the model)
- --multi – Indicates that Polygon fields should be created as MultiPolygonFields

- The output of the command may be copied directly into your models.py file for your application.

```
./manage.py ogrinspect --srid=2264 --mapping –-multi Wake_Open_Space.shp open_space
```

Sample use of the ogrinspect tool to generate a mapping and LayerMapping.

- Place the model that ogrinspect generates in your models.py file.
  - You'll likely need to modify the model – for example, fields are generated as requiring values, but your data might leave some fields blank.
  - Add index information as appropriate based on the methods that you use to locate data.
  - Change index/identifier fields as necessary.
- Save the LayerMapping in a separate file that you can later use to bulk-load your shape file features and attributed into your database.

Installing the Model in the Database

- Once you've create a model for each of your shapefiles, you can use the standard 'syncdb' argument to manage.py to create the model in the database:
  `./manage.py syncdb`
  - This command will push the model into the database, and issue the appropriate create table commands, etc.
- You can now connect to the database, and use the appropriate database commands to verify that the model has been created.

Remember, you'll need to add your new application to your settings.py INSTALLED_APPS section otherwise your app model won't be instantiated in the database.

```
                                Table "public.wake_fireresponse"
  Column   |           Type           |                              Modifiers
-----------+--------------------------+-----------------------------------------------------
 id        | integer                  | not null default nextval('wake_fireresponse_id_seq')
 district  | character varying(254)   | not null
 area      | double precision         | not null
 len       | double precision         | not null
 geom      | geometry                 | not null
Indexes:
    "wake_fireresponse_pkey" PRIMARY KEY, btree (id)
    "wake_fireresponse_geom_id" gist (geom)
Check constraints:
    "enforce_dims_geom" CHECK (ndims(geom) = 2)
    "enforce_geotype_geom" CHECK (geometrytype(geom) = 'MULTIPOLYGON'::text OR geom IS NULL)
    "enforce_srid_geom" CHECK (srid(geom) = 2264)
```

The PostgreSQL table that is created when the model defined in the previous example is installed in the database.

Importing Data Using the LayerMapping

1. Create a new instance of a LayerMapping object.

```
from django.contrib.gis.utils import LayerMapping      # Import LayerMapping
from wake.models import *                              # Import the models
lm = LayerMapping(model=FireResponse,                  # Create the Object
                  data='shapes/Wake_FireResponse_2008_12.shp',
                  mapping=mapping,
                  encoding='latin-1')
```

- The object needs to be passed in several named arguments (Required arguments are in bold, the remainder are optional. Recommended arguments are in italics):

Argument	Description
model	The geographic model that we are mapping into (not an instance of the model, as the mapper will create instances on its own)
data	The file to be imported (a string), or a reference to a DataSource instance.
mapping	The mapping dictionary created in the previous step.
layer	The index of the layer to use from the data source (used when the data source has multiple layers). Default is 0.

Argument	Description
`source_srs`	For Shapefiles (and other source files) without projection data, this may be used to specify the spatial reference system of the source data.
`encoding`	Used to specify the character encoding of the source file (for example, 'latin-1'). This is used to set the client encoding to ensure that the appropriate character set translations occur between the client and server.
`transaction_mode`	Either 'commit_on_success' or 'autocommit', indicates how data will be committed to the database. Default is 'commit_on_success' .
`transform`	If set to False, this will disable automatic coordinate system transformation. If set to false, invalid data may be placed in a table.
`unique`	If passed in with an argument of field names, the import will attempt to make those names unique by adding features with the same "keys" to a single feature set.

2. Use the save() method of the LayerMapping object to save the data into the database.
   `lm.save(verbose=True)`
   - The save() method accepts several named arguments:

Argument	Description
`fid_range`	When provided with a tuple that contains a begin and end value (begin, end), this will only import features within the specified range.
`progress`	When present, status information will be printed giving the number of features processed and save. This occurs for every 1000 features by default, or the value specified by progress (i.e., progress=100)
`silent`	Suppress any non-fatal error notifications.
`step`	Issue transaction commit statements every step interval (i.e., step=10 results in a commit; executed for every 10 features added)
`stream`	Accepts a file handle and writes status information to the handle (otherwise sys.stdout is used)
`strict`	Mapping will stop as soon as an error is encountered (default is to continue)
`verbose`	When set to True, information is printed for each record saved in the database.

GeoDjango Spatial Queries

About Spatial Lookups

- Unlike regular queries, which look for "matches", spatial queries rely on items such as distance and proximity.
  - These features are provided using a special set of query operators that are common across most spatial databases (known as the OGC Simple Features for SQL specification).
- GeoDjango extends the Django Object-relational model to add a number of spatial query methods.

Constructing Geometries

- Before we can run queries that answer questions such as "how many houses are within 1 mile of my house?" we need to generate a geometry that represents "my house".
- The GEOS set of libraries may be used to generate geometries, using a pre-defined set of object classes that represent geometries.
  - Each of these classes (used to generate geometries) accept as input a set of points and SRID information, and returns an instance of a GEOSGeometry object.
  - The GEOSGeometry object is then passed into a spatial query (by generating a GeoFilterQuerySet), and then may be executed against geometry information in the database.
- The table below describes the classes that may be used to generate GEOSGeometry objects:

```
from django.contrib.gis.geos import *
```

In order to generate GEOS objects, you must import the appropriate GEOS module components.

Class	Description
`Point()`	Accepts as arguments a set of X, Y, and Z coordinates (Z coordinates are only used for 3DZ geometries), and an SRID (in that order). Named arguments are 'x','y','z', and 'srid'.
`LineString()`	Generates a Line String. This may accept as arguments a list, tuple, or array. The values passed in should be x,y coordinate pairs an srid

Class	Description
	argument may also be provided.
`Polygon()`	Generates a Polygon. In this case, a tuple of tuples should be provided, indicating the geometry. This is essentially the same as providing WKT information for a polygon with a series of points, just without the WKT components.

- Some examples of creating new geometry objects are shown in the table below:

Example	Description

`from django.contrib.gis.geos import *`
`p = Point(2088767.336, 840283.137, 2264)`
`p2 = Point(x=2088767.336, y=840283.137, srid=2264)`
	Generate a GEOS Geometry object representing a point in SRID 2264.

`from django.contrib.gis.geos import *` `p2 = Point(2088767.336, 840283.137)`	The same as the previous example, but the SRID is undefined, and GeoDjango will be unable to perform an automatic coordinate system transformations using it.

`from django.contrib.gis.geos import *`
`l1 = LineString((2088767.336, 840283.137),`
`(2087840.178, 840139.596),`
`(2077773.000, 827907.187),`
	Generate a GEOS Geometry object representing a line (connect the dots (points) to form the line).

Example	Description
srid=2264)	
from django.contrib.gis.geos import * poly = Polygon(((2076922.6251, 827588.75012), (2076919.6962, 827581.6790), (2076912.6251, 827578.7501), (2076905.5541, 827581.6790), (2076902.6251, 827588.7501), (2076905.5541, 827595.8211), (2076912.6251, 827598.7501), (2076919.6962, 827595.8211), (2076922.6251, 827588.7501)), srid=2264)	Generates a Polygon object using a series of tuples, each which represents a single point in a linestring. Other (more advanced) polygons may be generated in a similar manner, and even "holes" can be carved out of a polygon by using a smaller, non-intersecting polygon inside a larger polygon.

- We'll use these geometries when we execute queries, and GeoDjango will automatically return a GEOS Geometry object when we retrieve data from a geometry field in the model.
  - This makes it easy to perform joins between geometry columns in tables, and also to generate spatial queries.

Actually, the database will return data in the Well-Known Text or Well-Known Binary format. GeoDjango will convert that to a GEOS Geometry object when you try to access it. This 'just in time' method saves work if a result row is never examined, and is known as a "lazy geometry".

Generating a GeoQuerySet

- Like Django's QuerySet, a GeoQuerySet is generated when we use GeoDjango components to build our models.
- GeoQuerySets and QuerySets are very similar, with the exception that GeoQuerySet's have a set of extra query methods that may be used to perform powerful spatial searching and matching in the database.
- In the table below we've described some of the common (but not all) query keywords that are used with a GeoQuerySet – these are analogous in many ways (and used in the same way) as methods in a QuerySet, with the obvious difference that the argument to a geometric operation with a GeoQuerySet is a GEOS Geometry object.

Operator	Description
`contains`	Tests whether the field contains the lookup geometry.
`coveredby`	Tests if no point in the geometry field is outside of the lookup geometry.
`covers`	Tests if no point in the lookup geometry is outside of the geometry field.
`crosses`	Tests if a geometry field spatially crosses the lookup geometry.

Operator	Description
`disjoint`	Tests is the lookup field is spatially disjoint from the geometry field.
`dwithin`	Tests if the field is within the specified distance of the lookup geometry. The parameter is a tuple, containing the lookup geometry, and the distance (either a Distance object, or a numeric value in units). This does not work for geographic coordinate systems (i.e., where degrees are used as the unit)
`equals`	Tests if the geometry field is spatially equal to the lookup geometry.
`intersects`	Tests if the geometry field and lookup geometry intersect.
`overlaps`	Tests is the geometry field spatially overlaps the lookup geometry (i.e., the overlap may be represented as a polygon).
`within`	Tests if the geometry field is spatially within the lookup geometry.
`distance_lte` `distance_gte`	These methods all accept a tuple as an argument (the first component being the geometry, and the second being the

Operator	Description

`distance_lt`
`distance_gt`
`dwithin`
	distance). The result is a GeoQuerySet that returns the number of objects within the distance specified.

- Like all Django models, you can use the geometry field (just as you can any other field type) in a filter:

```
fs = fireresponse.objects.filter(geom__contains=Point(2088767.336,
                                                      840283.137, srid=2264))
```

  - In the example above, we use a spatial query to determine the fire station that might respond to an emergency at the specified point. Note that reprojection will automatically occur if needed, since an SRID is provided.

- PostGIS also has a number of "lossy" lookups that (generally) focus only on index information (and are thus much faster).
  - These are query components such as "bbcontains" (test if a bounding box completely contains another bounding box), "left" and "right" (determine if a bounding box is to the left/right of another bounding box), etc.
  - These are documented in the GeoDjango documentation, but aren't covered in depth here (for time reasons.).

Displaying Map Data

About Map Data

- Once we have data inside the database, we need to be able to view the data using user maps (not just admin maps).
- In order to do this, we need to determine how we want to display the data:
  - Do we display the WKT representation of our data (hard to view by users)
  - Do we attempt to display the data using a web-mashup (such as OpenLayers)?
  - Do we simply provide a simple GeoJSON, KML, GML, or other interface into the data, so it may be consumed by other web services?
- GeoDjango is able to provide data using any of these methods, by creating standard Django views and generating the output data required.
  - In the case of OpenLayers, we probably need to split things into a few components – a set of view to provide a KML, GeoJSON, or other interface to streaming data to the client, and another view to actually display the basic data.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Generating Output Interfaces

- When we use tools like OpenLayers, Google Earth, etc. they expect to get layer data as an input stream.
- In order to provide this, there are a number of supported output formats, such as GML, KML, GeoJSON, and others.
  - GeoDjango supports KML, GML and GeoJSON output formats (just to name a few).
- In order to stream layer data, we'll need to generate some basic views that enable us to stream data out to a client.
  - Later, we can use these views to also provide more detailed information – such as query parameters, or a set of extents – to enable our application to only output useful data (as opposed to everything).

Certain data formats (such as KML) require a specific projection be used (KML uses EPSG:4326) . As a result, you will need to ensure that when you generate KML, the data you pass in has already been translated to the appropriate projection.

Using GeoDjango Shortcuts

- Like Django, GeoDjango provides several easy-to-use shortcuts to generate basic output.
- Shortcuts provided by GeoDjango include:
  - render_to_kml – Renders the response as KML, using the appropriate MIME types, etc.
  - render_to_kmz – Renders a compressed KML string and returns it as KMZ with the appropriate MIME type.
  - render_to_text – Renders the response using a plain text MIME type
- All these shortcuts do is take a regular template and template parameters, and render the template.
  - Template rendering would rely on the GeoQuerySet having a kml attribute, which is used to extract KML data.
- A sample KML template is available in 'site-packages/django/contrib/gis/templates/gis/kml'
  - You can copy these to your own 'gis/kml' template directory to customize them.
  - The default files are 'base.kml' and 'placemarks.kml'

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Generating KML Output

- You can now generate KML output using your data fairly easily.

```
from wake import models
from django.shortcuts import render_to_response
from django.contrib.gis.shortcuts import render_to_kml
# Create your views here.
def kml(request):
    fireresponse=models.fireresponse.objects.kml()
    return render_to_kml("gis/kml/placemarks.kml", {'places': fireresponse })
```

A sample view to generate KML output containing all of the fire response zones.

  - In the example shown here, we use render_to_kml to generate the output.
  - The placemarks.kml expects a 'places' template variable that contains the GeoQuerySet, which of course needs to have a kml attribute.
    - Placemarks.kml will use the name and description attributes of the object.
    - To use more than that, just create your own copy of the file in your template directory and use that file.
- Using render_to_kmz in the same fashion would result in compressed output.
- You can customize the KML output by modifying the 'placemarks.kml' file, this allowing you to have much more custom formatted data.

- If you examine the KML output, you'll notice that all the output uses the WGS84 SRS.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Integrating OpenLayers

About OpenLayers

- OpenLayers is a JavaScript-based library designed to display and manage geospatial data in the web browser.
- Written entirely in JavaScript, OpenLayers has no requirements for client workstations aside from a modern Web 2.0 capable web browser.
- OpenLayers supports many standard formats for data interchange over the web, including:
  - GeoJSON
  - Web Map Services (WMS)
  - Web Feature Services (WFS)
  - KML
  - GML
- OpenLayers has the ability to parse and draw vector as well as raster images, and can control transparency, image overlays, and a wide range of other commonly used techniques for displaying map data on the internet.

OpenLayers understands WMS and WFS services, so we can configure it to utilize our MapServer provided imagery via WMS or WFS. We can also use MapServer to control features that appear (or disappear) at different zoom levels (using MAX and MIN SCALE MAP elements) .

Likewise, we can generate "static" images for print or fax if we know the layers that are selected in OpenLayers, and the extents of the image. This gives us the ability to have a nice graphical interface for a user, and a nice way to print a map as it looks on screen (without using the browser).

Raster vs. Vector Data

- OpenLayers works by displaying a base layer of information (image) that is in raster format (image).
  - The base layer determines basic coordinate system information, and is typically provided by some web service - such as Google Maps, the OpenStreetMap project, Microsoft Virtual Server, Yahoo! Maps, USGS supplied Topographic Imagery, your own Ortho-imagery, or some other source data.
    - Data of this type is usually provided using some standard mechanism, such as a Web Map Service (WMS, a OGC standard), or via some supported proprietary method, such as the Google Maps API.
- As you've already seen, GeoDjango works primarily with vector data, and is generally unable to provide the underlying base layer data required for OpenLayers.
  - Typically, this layer comes from somewhere else.
  - Using tools like UMN MapServer, you can use the same PostGIS data to generate your own "base" map images - without relying on a service provider such as Google.
    - Often, third parties that provide map imagery may have limitations on use, and might change their license terms and conditions without prior notice to users.
- Once a raster base layer is available, we can use several techniques to provide lower-level map data using OpenLayers.

Installing OpenLayers

- Since OpenLayers is a pure JavaScript implementation, we need not install a large amount of server-side code in order to get it to work.
  - Download the OpenLayers JavaScript library at http://openlayers.org/download/ . There you can download the latest version of OpenLayers.
- Once downloaded, you need to unpack the Openlayers library and place in in a WWW accessible location.
  - On any pages where you wish to use Openlayers, you should add code similar to the following the the HEAD section of your HTML page(s):

```
<script src="/Openlayers-2.8/OpenLayers.js"></script>
```

    - This causes the browser to load the OpenLayer library components into the browser.
    - Once loaded, we can use OpenLayers calls to generate, manage, and display maps.
- Since OpenLayers is written in pure JavaScript, with no server-side components, it requires that a proxy server be available so that it can perform WMS and WFS requests against different servers.
  - This is because, for security reasons, JavaScript applications can only contact the server from which they originate.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Creating a New Map

- In OpenLayers, maps are generated and placed on a web page.
  - Each map is created using the 'OpenLayers.map' constructor.
    - That means if you want three maps displayed, you'll need to call three separate instances of the constructor.
  - When the object is first created, it has no layer data, so you'll need to call the appropriate methods to add layers to the existing object.
- When generating a new map, the constructor has a required argument – the name given to the map.
  - The name specified represents the ID for the map object, and should be distinct from other id tags on the same web page.
  - The OpenLayers map will retrieve the style settings for the DIV tag corresponding to the ID of the map, and size the map to match that size.
    - If you omit size information for the tag, the map will not be visible (it will be too small).
    - This method allows us to size maps anywhere in our page, with arbitrary sizes.
    - If the extents of the map are such that it will not fill the area, you'll find that the map is proportionally sized to fit a single dimension of the area.
      - As you zoom in, you'll find that the map fills the area.
- In addition to the name, we can specify an array of other map options, a subset of which is shown in the table below (note: this is not, by any means, a complete list of options):

Option	Description
`maxExtent`	A bounds object that indicates the extents of this map. This is specified by creating a bounds object that contains the extents of the map: `new OpenLayers.Bounds(-8793700, 4234437, -8710869, 4311265)`
`projection`	The projection of this map. Specified as an OpenLayers Projection object: `new OpenLayers.Projection("EPSG:900913")`
`units`	The units to use when displaying the layer. This, of course, will effect the scale-resolution. The units of Spherical Mercator are meters ('m')
`DisplayProjection`	When Proj4js support is enabled, this can be used to reproject underlying layers to the map projection. The value should be an instance of an OpenLayers.Projection object: `new OpenLayers.Projection('EPSG:2960')` When set, this projection will apply to any controls without a set projection as they are added to the map, controlling the units and other information that is displayed to the *user.*

- Note that this list of options isn't extensive, and these may be changed after the map is generated.

- Once you've determined what settings you require for your map, you can generate the map:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OpenLayers Example</title>
<!-- Style information for the “map” DIV is important -->
<style type="text/css">
  #map {
        width: 512px;
        height: 600px;
        border: 1px solid #eee;
      }
</style>
<script src="/OpenLayers-2.8/OpenLayers.js"></script>
<script type="text/javascript">
 var map = null;
 function OLinit() {
      var options = {
      'units' : "m",
      'numZoomLevels' : 15,
```

```
        'projection' : new OpenLayers.Projection("EPSG:900913"),
        'maxExtent' : new OpenLayers.Bounds(-8793700,
                                            4234437, -8710869, 4311265),
        'displayProjection': new OpenLayers.Projection("EPSG:4326"),
        };
        map = new OpenLayers.Map('map', options);
        map.zoomToMaxExtent();
</script>
</head>
<!-- Don't initialize the map until the body
     has completed loading -->
<body onload="init()">
      <div id="map"></div>
</body>
</html>
```

- The example above creates and loads a simple map with EPSG:900913 (aka Spherical Mercator) .
  - Note that the default SRS is EPSG:4326, both for the map object, and any underlying layer objects.
- With release 2.6, OpenLayers has the ability to perform reprojection using the Proj4JS JavaScript library, though at present it seems that only a limited set of projections are supported.

- For this reason, you should always provide your target EPSG value when requesting layers from WMS or WFS layers.
  - Note: WMS and WFS services are able to perform re projection as well.

Managing Layers

- One we've created our main map object, we can begin to add layers to it.
  - OpenLayers provides support for a wide range of different layer types, each which is added by creating a new layer object, and then adding the layer object to the map itself.
- Each layer type has its own constructor that we can call to create and manage the new layer.
  - This mechanism allows us to have a map that gets layers from a set of disparate locations, and then display them as layers all on the same OpenLayers map.
  - Layers can be managed, displayed, hidden, and changed via JavaScript through the OpenLayers API.
- To add a new layer, we need to locate the type of layer we wish to display and create an object for it.
  - A short list of layer types is listed in the table below:

Layer Type	Description
`Boxes`	The boxes layer generates a set of DIV tags that are overlayed on top of a map. This gives the impression of boxes drawn on the map.
`GML`	Creates a new layer (vector) by parsing the contents of a Geography Markup Language (GML) file. GML is an OGC standard.
`Google`	Used to add a google layer to a map (note: a Google Maps API key is required to use Google Maps)

Layer Type	Description
`Grid`	Used to create a Grid layer for a map.
`KaMap`	Used to add a KaMap base layer to a map.
`MapServer`	Used to use the MapServer CGI parameters to use MapServer (Note: MapServer can also be used using WMS or WFS, if configured to provide either)
`Markers`	Used to create a layer of markers.
`VirtualEarth`	Used to add a Microsoft Virtual Earth layer to a map.
`WFS`	Used to add a Web Feature Service layer to a map.
`WMS`	User to add a Web Map Service layer to a map.
`Yahoo`	Used to add a Yahoo! Map layer to a map.
`Text`	Used to add text data to a map.

- Each of the different layer objects is generated by calling the constructor for the object, with each constructor method having a different set of arguments – based on the layer to be generated.
  - Constructors are generally of the form OpenLayers.Layers.[name], where [name] is replaced by the layer type:

```
var wms = new OpenLayers.Layers.WMS();
```

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

```
var gmap = new OpenLayers.Layers.Google()
var MapServer = new OpenLayers.Layers.MapServer()
```

- Note that each of these methods has its own set of required arguments to instantiate the layer.

- Whenever we add a layer to the map, we first create the object to represent the layer, and then call the addLayer() method of the map object, passing in the layer object we just created.

OpenLayers Base Layer Requirement

- Whenever we display map data in OpenLayers, we are required to provide at least a single non-transparent ("base") layer.
  - The base layer is used when determining various map related information, such as bounding information, and information about mouse position, etc.
- Layers that are "base" layers have their "transparent" option set to false.
  - OpenLayers uses the "transparent: false" option to show the base layer as a base layer in the LayerSwitcher control (more about this layer).
  - An alternative to using the LayerSwitcher control is to enable and disable the layer directly through OpenLayers.

```
<script type="text/javascript"
        src="http://openstreetmap.org/openlayers/OpenStreetMap.js">
</script>
<script type="text/javascript">
var osm = new OpenLayers.Layer.OSM.Mapnik("OpenStreetMap");
</script>
```

A sample base layer using OpenStreetMap.

- We can have multiple base layers, but only a single base layer will be drawn at a time.

Adding WMS & WFS Layers to a Map

- Adding new WMS or WFS layer requires that we create a layer object, using the OpenLayers.Layer.WMS() or OpenLayers.Layer.WFS() methods.
  - The method accepts up to 4 arguments:
    - The name of the layer (used for display purposes)
    - The base URL for the layer (note: if the layer is not on the same server as the HTML file, you'll need to have a proxy configured).
    - An object ({}) containing key/value pairs representing the GetMap query string parameters and values.
      - This allows us to pass additional arguments/settings to the GetMap request against the WMS service.
    - A hash of layer options to apply to the new layer.

Although OpenLayers supports reprojection of Vector data, it cannot reproject Raster data. For that reason, whenever using WMS services, we need to ensure that the WMS services that we utilize return data in the same projection as the base layer. This restriction does not exist for Vector data (such as WFS services, or GML data files).

```
var wake_property = new OpenLayers.Layer.WMS(
    "Wake County Property Data",
    "http://nibbler.otg-nc.com/cgi-bin/wake?",
    { layers: 'wake_property',
      transparent: true },
    { isBaseLayer: false,
      opacity: 0.5 }
);
map.addLayer(wake_property);
```

Sample JavaScript code to add a WMS layer to an existing map. The transparency of the layer indicates it is an overlay, and not a base layer.

- This would include information such as the projection of the layer, extents, units, scales, resolutions, etc.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

GeoDjango and OpenLayers

- GeoDjango provides data using a wide range of OpenLayers supported formats, including GML, GeoJSON, and KML.
  - These formats are typically requested using a single request, so passing in data using these formats can be cumbersome – especially since the viewport of the current map might be small in relation to the source data.
- In order to overcome some of these limitations, there are several things we can do to cause OpenLayers to request data in a more incremental manner.
  - We can cause OpenLayers to make WFS type requests (with a Bounding Box) to GeoDjango.
    - GeoDjango, in turn, can use the bounding information to generate KML output that contains only the features in the requested viewport.
    - This means that GeoDjango operates much like a KML server.
  - We can use a WMS service to generate a "raster" image that contains a representation of the requested layer, and then use OpenLayers GetFeature control to make specific requests to GeoDjango for additional data.
  - We can display a map and a small set of data, and provide a JavaScript interface to "advance" to another set of records (i.e., LIMIT 10, LIMIT 10 OFFSET 10, LIMIT 10 OFFSET 20)

Many of the techniques discussed here may also be implemented using a variety of other techniques and technologies.

Limiting Features Returned with GeoDjango

- It's possible for OpenLayers to generate queries that include GET parameters with a BBOX key that contains a set of 4 points that make up the bounding box around the current viewport.
- From this information, we can create a Polygon that is represented by these points.
- We can then use this with a simple 'bboverlaps' query to return a limited set of data.

In this tutorial we're using GeoDjango to provide the same services that could (arguably, more easily) be provided using a GetFeature Request. However, GeoDjango can take things a step further, such as limiting information returned based on a user being logged in, or using specially granted permissions.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

- This technique can be used with a wide range of different layers to provide "on the fly" generation of localized KML data.

- In order to utilize such a service, OpenLayers would use a layer definition such as the one shown here.

```
var wake_schools = new OpenLayers.Layer.WFS("Schools", "/wake_school/kml/", {},
                     {  projection: new OpenLayers.Projection("EPSG:4326"),
                        format: OpenLayers.Format.KML,
                        formatOptions: { extractAttributes: true },
                        styleMap: new OpenLayers.StyleMap({'externalGraphic': '${graphic}',
                                                           'pointRadius': 15})
                     } );
map.addLayer(wake_schools);
```

Sample OpenLayers code to add a WFS layer to our map using GeoDjango KML output. In this case, the externalGraphic indicates the image file that should be used as an icon for the feature, and the use of '${graphic}' indicates that the URL for the image should come from the KML data. The use of OpenLayers.Format.KML indicates that OpenLayers should expect (and parse) a KML response.

```
layers={ 'wake_school':   { 'model': models.school,
                            'maxfeatures':30 }  }
# The urlpattern for this might be (r'^(?P<layer>[^/]+)/kml/$', wake.views.kml)
def kml(request, layer=None):
    if layer not in layers: raise Http404 # Only allowed layers
    try:
        bbox=request.GET.get('BBOX', request.GET.get('bbox')) # Get the BBOX from the request
    except:
        raise Http404            # no bbox? No page for you!
    try:
        minx,miny,maxx,maxy=[float(i) for i in bbox.split(',')]  # get the coordinates...
        geom=Polygon(((minx,miny),(minx,maxy),(maxx,maxy),(maxx,miny),(minx,miny)),srid=4326)
    except:       # Lots of different errors could occur...handle
        raise Http404
    rescount=layers[layer]['model'].objects.filter(geom__bboverlaps=geom).count()
    if rescount < layers[layer]['maxfeatures']:   # Limit to 1000 features max
        res=layers[layer]['model'].objects.filter(geom__bboverlaps=geom).kml() # Locate Data..
        layer='placemarks'
        return render_to_kml('gis/kml/%s.kml' % layer, {'geomdata': res, 'fields': fields,
                                                        'places': res} )
    else:                 # too many features to display for this layer
        return render_to_kml('gis/kml/%s.kml' % layer, {})
```

A sample GeoDjango view to process a request for school location data.

Using the SelectFeature Control

- One of the more useful controls implemented by OpenLayers is the ability to use the SelectFeature control.
  - The SelectFeature control allows us to take point-clicks on vector layers and call events driven by those selects.
- The SelectFeature control allows for hover events (something happens when the mouse hovers over a feature), and the ability to select multiple geometries.

The SelectFeature control will work on any layer type that inherits from OpenLayers.Layer.Vector.

- To utilize the selectFeature control, we'll need to create a new instance of the control, passing in a layer that the control should be attached to.
- Once created, we can use the 'activate()' method to activate the control, and the 'deactivate()' method to start and stop the control.

```
var wake_schools_control = new OpenLayers.Control.SelectFeature(wake_schools);
map.addControl(wake_schools_control);
wake_schools_control.activate();
function wake_schools_onPopupClose(evt) { wake_schools_control.unselect(this.feature);  }
function wake_schools_select(evt) {
   feature = evt.feature;
   popup = new OpenLayers.Popup.FramedCloud(this.name + " Popup",
                                            feature.geometry.getBounds().getCenterLonLat(),
                                            new OpenLayers.Size(100,100),
```

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

```
                                                "<h2>"+feature.data.namelong + "</h2>",
                                                null,
                                                true,
                                                wake_schools_onPopupClose);
    feature.popup = popup;
    popup.feature= feature;
    map.addPopup(popup);
}
function wake_schools_unselect(evt) {
    feature = evt.feature;
    if (feature.popup) {
        popup.feature = null;
        map.removePopup(feature.popup);
        feature.popup.destroy();
        feature.popup=null;
    }
}
wake_schools.events.on({ 'featureselected': wake_schools_select,
                         'featureunselected': wake_schools_unselect });
```

- In the example shown here, a SelectFeature Control is added to the "wake_schools" layer.
- When the mouse "clicks" on a feature, the onFeatureSelect method is called to indicate the feature has been selected.
- When the mouse "clicks away" from a feature, the "on FeatureUnSelect method is called to

indicate the feature has been unselected.

- These callback functions can be used to generate popups, remove the feature, or perform other layer operations.

Making Feature Requests with GetFeature

- OpenLayers provides a GetFeature request method that may be used to translate a "click" on a map to an AJAX query.
  - This allows GeoDjango to respond simple clicks on a map, rather than having to provide a wide range of data.
  - The GetFeature method also allows GeoDjango to perform searches on an ad-hoc basis, and send the responses back to OpenLayers.
- In order to use GetFeature, we first need to create a control for it:

```
var protocol = new OpenLayers.Protocol.HTTP({ format: new OpenLayers.Format.KML(),
                                              url: '/wake_property/kml/feature/' });
var pselect = new OpenLayers.Control.GetFeature({ protocol: protocol,
                                                  click: true,
                                                  hover: false });
```

  - In the example above, the protocol object indicates that the HTTP protocol should be used to make the request.
  - The format of the response is KML, which is always returned in EPSG:4326 (that means that we'll need to reproject the response to match our EPSG:900913 projection).
- Once we've created the control, we'll need to create a layer to store the result.
  - In this case, a click may be turned into a feature that is drawn on the map using an OpenLayers

vector layer, so we need to create the layer and add it to our map:

```
var select = new OpenLayers.Layer.Vector("Selection", {displayInLayerSwitcher:
false});
map.addLayers(maplayers.concat(select));
```

- Lastly, we need to create functions that are called whenever a click occurs on a feature:

```
wgs84=new OpenLayers.Projection("EPSG:4326")
pselect.events.register("featureselected", this, function(e) {
   e.feature.geometry.transform(wgs84,map.projection)
   select.addFeatures([e.feature]);
   $("#featuredata").css("display", "");
   var rows="";
   for (var v in e.feature.data) {
     rows += "<tr class=\"row\"><td>" + v.replace(/_/g,' ') +
             "</td><td>" + e.feature.data[v] + "</td></tr>";
   }
   $("#featuretable").append(rows);
   });

pselect.events.register("featureunselected", this, function(e) {
   select.removeFeatures([e.feature]);
   $("#featuredata").css("display", "none");
   $("#featuretable .row").remove();
```

```
});
```

- In the code above, we also use a short for loop to populate a table with attribute information.
- The addFeature and removeFeature methods of the 'select' layer allow us to add/remove objects from the displayed layer one at a time.

- The final step is to add the control to the map and activate it:

```
map.addControl(pselect);
pselect.activate();
```

- Once this is done, we can re-load the map and verify proper operation.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Questions?

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Where to Get Answers/Next Steps

- Take our Python, Django, GeoDjango, PostgreSQL, PostGIS or other training: **http://www.opentechnologygroup.com**
- Python.org is the central site for development and documentation of Python.
  - Comprehensive documentation, along with working examples, is there for you to read and review.
- O'Reilly and Associates offers several Python books, including:
  - Learning Python
  - Python Cookbook
  - Programming Python
- Of these books, we recommended the Cookbook and Programming Python as next steps in better understanding Python.
- OTG offers a wide range of advanced courses related to Python, including Django, GeoDjango, Advanced Python training, and more. Contact us for more information - or for custom training needs.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com