django-geoip Documentation

Release 0.2.4

coagulant

April 15, 2012

CONTENTS

App to figure out where your visitors are from by their IP address.

Detects country, region and city, querying the database with geodata. Optional *high-level API* provides user location in request object.

Note: Currentrly `django-geoip` supports only ipgeobase.ru backend. It provides accurate geolocation in Russia and Ukraine only. There are plans to add other backends in future releases.

CHAPTER

ONE

CONTENTS

1.1 Installation

This app works with python 2.6-2.7, Django 1.3 and higher.

Recommended way to install is via pip:

```
pip install django-geoip
```

1.1.1 Basic

- Add `django_geoip` to `INSTALLED_APPS` in settings.py:

  ```
  INSTALLED_APPS = (...
                    'django_geoip',
                    ...
                   )
  ```

- Create application tables in database:

  ```
  python manage.py syncdb
  ```

  If you're using South:

  ```
  python manage.py migrate
  ```

- Obtain latest data to perform geoip detection by *running management command*:

  ```
  python manage.py geoip_update
  ```

1.1.2 Advanced

In order to make *user's location detection automatic* several other steps are required:

- Add `LocationMiddleware` to `MIDDLEWARE_CLASSES`:

  ```
  MIDDLEWARE_CLASSES = (...
      'django_geoip.middleware.LocationMiddleware',
      ...
  )
  ```

- Provide a custom *location model* (inherited from `django_geoip.models.GeoLocationFascade`)

- Specify this model in *Settings*:

  ```
  GEOIP_LOCATION_MODEL = 'example.models.Location' # just an example, replace with your own
  ```

- Include app urls into your urlconf if you want to *allow visitors to change their location*:

  ```
  urlpatterns += patterns('',
      ...
      (r'^geoip/', include('django_geoip.urls')),
      ...
  )
  ```

1.2 Usage

The app provides both high and low-level APIs to work with geolocation. Low-level API works super-simple: it guesses geographic location by an IP adress. High-level API is more complex and deeply integrated in Django: it automatically detects user location in every request and makes it available as `request.location`.

1.2.1 Low-level API usage

Low-level API allows you to guess user's location by his IP address. This function returns a *database record*, associated with IP's city, region and country.

Here is a basic example:

```
from django_geoip.models import IpRange

ip = "212.49.98.48"

try:
    geoip_record = IpRange.objects.by_ip(ip)

    print geoip_record.city
    # >>   ( )

    print geoip_record.region
    # >>    ()

    print geoip_record.country
    # >> Russia ()

except IpRange.DoesNotExist:
    print 'Unknown location'
```

1.2.2 High-level API usage

The app provides a convenient way to detect user location automatically. If you've followed *advanced installation instructions*, user's location should be accessible via `request` object:

```
def my_view(request):
    """ Passing location into template """
    ...
    context['location] = request.location
    ...
```

`request.location` is an instance of a custom model that you're required to create on your own (details below).

Location model rationale

Location model suites the basic needs for sites with different content for users, depending on their location. Ipgeobase forces Country-Region-City geo-hierarchy, but it's usually too general and not sufficient. Site content might depend on city only, or vary on custom areas, combining various cities, that don't match actual geographic regions.

In order to abstract geography from business logic, *django-geoip requires a model*, specific to your own app.

Creating custom location model

Create a django model, that inherits from `django_geoip.models.GeoLocationFascade`. It might be a proxy model that doesn't require a separate database table, but it might be handy in many cases.

Location should implement following classmethods:

`get_available_locations``()`
Returns a queryset of all active custom locations.

`get_by_ip_range``(`*ip_range*`)`
Returns single instance of location model, corresponding to specified ip_range. Raises `DoesNotExist` if no location is associated with give IP address.

`get_default_location``()`
Returns single instance of location model, acting as a fallback when `get_by_ip_range` fails.

Example of custom location model

Very basic implementation of `GeoLocationFascade` for demonstration purpose:

```
class MyCustomLocation(GeoLocationFascade):
    """ Location is almost equivalent of geographic City.
        Major difference is that only locations
        from this model are returned by high-level API, so you can
        narrow down the list of cities you wish to display on your site.
    """
    name = models.CharField(max_length=100)
    city = models.OneToOneField(City, related_name='my_custom_location')
    is_default = models.BooleanField(default=False)

    @classmethod
    def get_by_ip_range(cls, ip_range):
        """ IpRange has one to many relationship with Country, Region and City.
            Here we exploit the later relationship."""
        return ip_range.city.my_custom_location

    @classmethod
    def get_default_location(cls):
        return cls.objects.get(is_default=True)

    @classmethod
    def get_available_locations(cls):
        return cls.objects.all()
```

1.2.3 Switching user's location

Switching location from front-end is very much like changing language in Django (in fact the code is almost the same with a little bit of difference, docs are a nice rip-off).

> As a convenience, the app comes with a view, `django_geoip.views.set_location`, that sets a user's location and redirects back to the previous page.
>
> Activate this view by adding the following line to your URLconf:
>
> ```
> # Note that this example makes the view available at /geoip/change/
> (r'^geoip/', include('django_geoip.urls')),
> ```
>
> The view expects to be called via the POST method, with a location identifier `location_id` set in request. It saves the location choice in a cookie that is by default named `geoip_location_id`. (The name can be changed through the `GEOIP_COOKIE_NAME` setting.)
>
> After setting the language choice, Django redirects the user, following this algorithm:
>
> - Django looks for a `next` parameter in the POST data.
> - If that doesn't exist, or is empty, Django tries the URL in the `Referrer` header.
> - If that's empty – say, if a user's browser suppresses that header – then the user will be redirected to / (the site root) as a fallback.
>
> Here's example part of a view rendering a form to change location:
>
> ```
> def get_context(self, **kwargs):
>     return {'LOCATIONS': location_model.get_available_locations()}
> ```
>
> Here's example HTML template code:
>
> ```
> {% load url from future %}
>
> <form action="{% url 'geoip_change_location' %}" method="post">
> <input name="next" type="hidden" value="/next/page/" />
>     <select name="location_id">
>     {% for location in LOCATIONS %}
>     <option value="{{ location.id }}">{{ location.name }}</option>
>     {% endfor %}
> </select>
> <input type="submit" value="Change" />
> </form>
> ```

1.3 Under the hood

1.3.1 Data storage

All geoip data, including geograpy and geoip mapping is stored in the database. To avoid unnecessary database hits user location id is stored in a cookie.

Geography

Right now django-geoip supports only ipgeobase geography, which consist of following entities: Country, Region, City. Database maintains normalized relationships between all entities, i.e. Country has many Regions, Region has many Cities.

class `django_geoip.models.`**`Country`**`(`**args*, ***kwargs*`)`
One country per row, contains country code and country name.

class `django_geoip.models.`**`Region`**`(`**args*, ***kwargs*`)`
Region is a some geographical entity that belongs to one Country, Cities belong to one specific Region. Identified by country and name.

class `django_geoip.models.`**`City`**`(`**args*, ***kwargs*`)`
Geopoint that belongs to the Region and Country. Identified by name and region. Contains additional latitude/longitude info.

IP ranges

class `django_geoip.models.`**`IpRange`**`(`**args*, ***kwargs*`)`
IP ranges are stored in separate table, one row for each ip range.

Each range might be associated with either country (for IP ranges outside of Russia and Ukraine) or country, region and city together.

Ip range borders are stored as long integers

1.3.2 Backends

There is currently no infrastructure to use alternative geoip backends, but it's planned for future releases.

Ipgeobase backend

ipgeobase.ru is a database of russian and ukranian IP networks mapped to geographical locations.

It's maintained by RuCenter and updated daily.

As of 9 April 2012 it contains info on 952 cities and 145736 Ip Ranges (some networks doesn't belong to CIS).

Here a is demo of ip detection: http://ipgeobase.ru/

1.4 Updating GeoIP database

Note: Currentrly `django-geoip` supports only ipgeobase.ru backend.

To update your database with fresh entries (adds new geography and completely replaces all IpRegions with fresh ones):

```
python manage.py geoip_update
```

If you wish to clear all geodata prior the sync (deletes all Cities, Regions, Countries and IpRanges):

```
python manage.py geoip_update --clear
```

1.5 Settings

`django-geoip` has some public configuration:

class `django_geoip.geoip_settings.`**`GeoIpConfig`**`(`*`**kwargs`*`)`

GeoIP configuration

`COOKIE_DOMAIN` **= ‘’**

Cookie domain for LocationCookieStorage class.

`COOKIE_EXPIRES` **= 31622400**

Cookie lifetime in seconds (1 year by default) for LocationCookieStorage class.

`COOKIE_NAME` **= ‘geoip_location_id’**

Cookie name for LocationCookieStorage class (stores *custom location’s* primary key).

`LOCATION_MODEL` **= ‘django_geoip.models.GeoLocationFascade’**

A reference to a *model* that stores custom geography, specific to application.

`STORAGE_CLASS` **= ‘django_geoip.storage.LocationCookieStorage’**

Persistent storage class for user location

1.6 Reference

This section comtains documentation to module internals, useful for `django-geoip` developers.

1.6.1 GeoLocationFascade

class `django_geoip.models.`**`GeoLocationFascade`**`(`*`*args`*`,` *`**kwargs`*`)`

Interface for custom geographic models. Model represents a fascade pattern for concrete GeoIP models.

classmethod **`get_available_locations`**`()`

Return all locations available for users to select in frontend

Returns GeoLocationFascade

classmethod **`get_by_ip_range`**`(`*`ip_range`*`)`

Return single model instance for given IP range. If no location matches the range, raises DoesNotExist exception.

Parameters **ip_range** (*IpRange*) – User’s IpRange to search for.

Returns GeoLocationFascade single object

classmethod **`get_default_location`**`()`

Return default location for cases where ip geolocation fails.

Returns GeoLocationFascade

1.6.2 Locator

class `django_geoip.base.`**`Locator`**`(`*`request`*`)`

A helper class that automates user location detection.

`is_store_empty``()`

Check whether user location will be detected by ip or fetched from storage.

Useful for integration with *django-hosts*.

`locate``()`

Find out what is user location (either from his IP or cookie).

Returns *Custom location model*

1.6.3 location_model

`django_geoip.base.location_model` – SimpleLazyObject to get current *location model*.

1.7 Integrating with django-hosts

Django-hosts routes requests for specific hosts to different URL schemes defined in modules called "hostconfs". Django-geoip plays nice with django-hosts, allowing to redirect user to specific geodomain.

In this example `www.site.com` will redirect to `asia.site.com` for users from the East and `us.site.com` for american ones. For european users in will remain `www.site.com`` without redirect (default location).

0. *Install and setup django-geoip* Let's assume we have defined this custom location model:

   ```
   # app/models.py
   from django_geoip.models import Country, GeoLocationFascade

   class Location(GeoLocationFascade):
       slug = models.SlugField('Site kwarg')
       country = model.ForeignKey(Country)
   ```

   This model is referenced in `settings.py`:

   ```
   GEOIP_LOCATION_MODEL = 'app.models.Location'
   ```

   We also need to change default user location storage mechanism, because it's fully determined by hostname:

   ```
   GEOIP_STORAGE_CLASS = 'django_geoip.storage.LocationDummyStorage'
   ```

1. Install and setup django-hosts

   ```
   pip install django-hosts==0.4.2
   ```

   Make sure you also followed other steps: adding to INSTALLED_APPS, adding a middleware, creating `hosts.py`, setting up ROOT_HOSTCONF and DEFAULT_HOST.

   Note: `django_geoip.middleware.LocationMiddleware` should come before `django_hosts.middleware.HostsMiddleware` in `MIDDLEWARE_CLASSES` to make things work together.

2. Configure `host_patterns` in *hosts.py*:

   ```
   host_patterns = patterns('',
       # Default www.sitename.com pattern that redirects users to <location>.sitename.com
       # depending on their IP address
       host(r'www', settings.ROOT_URLCONF, name='www', callback=detect_location),

       # Geodomain for specific region: <location>.sitename.com, doesn't redirect
       host(r'(?P<site_slug>[\w-]+)', settings.ROOT_URLCONF, name='location', callback=save_locatic
   )
   ```

3. Define `detect_location` callback:

   ```
   from django_geoip.base import location_model, Locator
   from django_hosts.reverse import reverse_host

   def detect_location(request):
   ```

```
    """ Callback takes request object and redirects to specific location domain if appropriate "

    default_location = location_model.get_default_location()

    # User is a first-timer and doesn't have a cookie with detected location
    if Locator(request).is_store_empty():
        # If we're at www-address, but not from default location, then do redirect.
        if request.location != default_location:
            return _redirect(request, domain=reverse_host('location', kwargs={'site_slug': reque
    request.location = default_location
```

4. Define `save_location` callback:

```
def save_location(request, site_slug):
    """ Store location in request, overriding geoip detection """
    request.location = get_object_or_404(Location, slug=site_slug)
```

1.8 Changelog

1.8.1 0.2.4 (2012-04-15)

- Proper datamigration for countrynames
- `GeoLocationFascade` defines abstract classmethods
- `bulk_create` support for Django 1.4
- Default view url renamed from `change` to `setlocation`
- Improved docs a lot more
- Short tutorial for django-hosts integration

1.8.2 0.2.3 (2012-04-11)

- Added country names
- Management update command renamed from `ipgeobase_update` to `geoip_update`
- Management command verbose output with progressbar
- Dropped django 1.2 support
- Documentation improved

1.8.3 0.2.2 (2012-01-25)

- Fixed middleware behavior when `process_request` never ran (redirects)
- Improved location storage validation, fixed cookie domain detection
- Added `Locator.is_store_empty` function to reveal if geoip detection was made

1.8.4 0.2.1 (2012-01-25)

- Fixed middleware behavior when request.location is None
- Added `GEOIP_STORAGE_CLASS` setting to override default user location storage
- Introduced `LocationDummyStorage` class to avoid cookie storage

1.8.5 0.2 (2012-01-20)

- Major refactoring of the app, added more tests
- Fixed a typo in `get_availabe_locations`

1.8.6 0.1 (2012-01-18)

- Initial release

CHAPTER

TWO

DEVELOPMENT

You can grab latest code on dev branch at Github.

Feel free to submit issues, pull requests are also welcome.

CHAPTER

THREE

TESTS

3.1 Running tests

This app uses nose test framework and django_nose.

Most of the `django-geoip` code is covered with tests, so if you wish to contribute (which is highly appreciated) pelase add corresponging tests.

Running tests:

```
python runtests.py
```

This command rund only unittests, which is preferred behavir. However, you might need to run system tests too (might take some time and requires internet connection):

```
python runtests -c system.cfg
```

You can run testsuite this way:

```
python manage.py runtests.py
```

More on tests

PYTHON MODULE INDEX