Бесплатная электронная книга

УЧУСЬ Django

Free unaffiliated eBook created from
Stack Overflow contributors.

#django

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django

It is an unofficial and free Django ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Django.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Django

замечания

Django рекламирует себя как «веб-инфраструктуру для перфекционистов с предельными сроками», а «Django упрощает создание лучших веб-приложений быстрее и с меньшим количеством кода». Его можно рассматривать как архитектуру MVC. В его основе он имеет:

- легкий и автономный веб-сервер для разработки и тестирования
- система сериализации и проверки формы, которая может переводить между HTML-формами и значениями, подходящими для хранения в базе данных
- система шаблонов, использующая концепцию наследования, заимствованную из объектно-ориентированного программирования
- структура кэширования, которая может использовать любой из нескольких методов кеширования для классов промежуточного программного обеспечения, которые могут вмешиваться на разных этапах обработки запроса и выполнять пользовательские функции
- внутренняя диспетчерская система, которая позволяет компонентам приложения передавать события друг другу через заранее определенные сигналы
- система интернационализации, включая перевод собственных компонентов Django на различные языки
- система сериализации, которая может создавать и читать XML и / или JSON-представления экземпляров модели Django
- система для расширения возможностей механизма шаблонов
- интерфейс к встроенной платформе тестирования Python

Версии

Версия	Дата выхода
1,11	2017-04-04
1,10	2016-08-01
1,9	2015-12-01
1,8	2015-04-01
1,7	2014-09-02
1,6	2013-11-06

Версия	Дата выхода
1,5	2013-02-26
1.4	2012-03-23
1,3	2011-03-23
1.2	2010-05-17
1,1	2009-07-29
1,0	2008-09-03

Examples

Запуск проекта

Django - это основа веб-разработки на основе Python. Django **1.11** (последняя стабильная версия) требует установки Python **2.7** , **3.4** , **3.5** или **3.6** . Предполагая, что `pip` доступен, установка выполняется так же просто, как выполнение следующей команды. Имейте в виду, что, опуская версию, как показано ниже, будет установлена последняя версия django:

```
$ pip install django
```

Для установки конкретной версии django предположим, что версия django **1.10.5** , выполните следующую команду:

```
$ pip install django==1.10.5
```

Веб-приложения, созданные с использованием Django, должны находиться в проекте Django. Вы можете использовать команду `django-admin` для запуска нового проекта в текущем каталоге:

```
$ django-admin startproject myproject
```

где `myproject` - это имя, которое однозначно идентифицирует проект и может состоять из **цифр** , **букв** и **подчеркиваний** .

Это создаст следующую структуру проекта:

```
myproject/
    manage.py
    myproject/
        __init__.py
        settings.py
```

```
        urls.py
        wsgi.py
```

Чтобы запустить приложение, запустите сервер разработки

```
$ cd myproject
$ python manage.py runserver
```

Теперь, когда сервер работает, посетите `http://127.0.0.1:8000/` с помощью вашего веб-браузера. Вы увидите следующую страницу:

По умолчанию команда `runserver` запускает сервер разработки на внутреннем IP- `runserver` на порту `8000` . Этот сервер будет автоматически перезагружаться при внесении изменений в ваш код. Но если вы добавите новые файлы, вам придется вручную перезапустить сервер.

Если вы хотите изменить порт сервера, передайте его как аргумент командной строки.

```
$ python manage.py runserver 8080
```

Если вы хотите изменить IP-адрес сервера, передайте его вместе с портом.

```
$ python manage.py runserver 0.0.0.0:8000
```

Обратите внимание, что `runserver` работает только для отладки и локального тестирования. Специализированные серверные программы (такие как Apache) всегда должны использоваться в производстве.

Добавление приложения Django

Проект Django обычно содержит несколько `apps` . Это просто способ структурирования вашего проекта в небольших поддерживаемых модулях. Чтобы создать приложение, перейдите в свою папку проекта (где находится файл `manage.py`) и запустите команду `startapp` (измените *myapp* на все, что вы хотите):

```
python manage.py startapp myapp
```

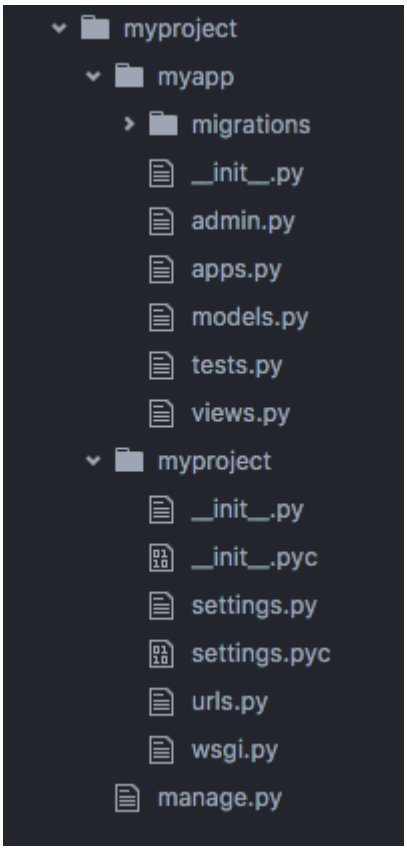
Это создаст для вас папку *myapp* и некоторые необходимые файлы, например `models.py` и `views.py` .

Чтобы Django узнал о *myapp* , добавьте его в свой файл `settings.py` :

```
# myproject/settings.py

# Application definition
INSTALLED_APPS = [
    ...
    'myapp',
]
```

Папка-структура проекта Django может быть изменена в соответствии с вашими предпочтениями. Иногда папка проекта переименовывается в `/src` чтобы избежать повторения имен папок. Типичная структура папок выглядит так:



Концепции Django

django-admin - это инструмент командной строки, который поставляется с Django. Он поставляется с несколькими полезными командами для начала и управления проектом Django. Команда такая же, как `./manage.py` , с той разницей, что вам не нужно находиться в каталоге проекта. `DJANGO_SETTINGS_MODULE` быть установлена переменная среды `DJANGO_SETTINGS_MODULE` .

Проект **Django** представляет собой кодовую базу Python, содержащую файл настроек Django. Проект может быть создан администратором Django с помощью команды `django-admin startproject NAME` . Обычно проект имеет файл `manage.py` на верхнем уровне и файл корневого URL-адреса, называемый `urls.py` `manage.py` - это версия `django-admin` конкретного проекта и позволяет запускать команды управления в этом проекте. Например, чтобы запустить проект локально, используйте `python manage.py runserver` . Проект состоит из приложений Django.

Приложение Django представляет собой пакет Python, содержащий файл моделей (по умолчанию - `models.py`) и другие файлы, такие как URL-адреса приложений и представления. Приложение можно создать с помощью команды `django-admin startapp NAME` (эта команда должна запускаться из вашей директории проекта). Чтобы приложение было частью проекта, оно должно быть включено в список `INSTALLED_APPS` в `settings.py` . Если вы использовали стандартную конфигурацию, Django поставляется с несколькими приложениями своих собственных предустановленных приложений, которые будут обрабатывать вас как аутентификацию . Приложения могут использоваться в нескольких проектах Django.

Django ORM собирает все модели баз данных, определенные в `models.py` и создает таблицы базы данных на основе этих классов моделей. Для этого сначала настройте свою базу данных, изменив параметр `DATABASES` в `settings.py` . Затем, как только вы определили свои модели баз данных , запустите `python manage.py makemigrations` за которым следует `python manage.py migrate` чтобы создать или обновить схему вашей базы данных на основе ваших моделей.

Полный пример приветствия.

Шаг 1 Если у вас уже установлен Django, вы можете пропустить этот шаг.

```
pip install Django
```

Шаг 2 Создайте новый проект

```
django-admin startproject hello
```

Это создаст папку с именем `hello` которая будет содержать следующие файлы:

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Шаг 3 Внутри модуля `hello` (папка, содержащая `__init.py__`) создайте файл с именем

`views.py` :

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── views.py  <- here
│   └── wsgi.py
└── manage.py
```

и добавьте следующее содержание:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('Hello, World')
```

Это называется функцией просмотра.

Шаг 4 Отредактируйте `hello/urls.py` следующим образом:

```
from django.conf.urls import url
from django.contrib import admin
from hello import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.hello)
]
```

который связывает функцию вида `hello()` с URL-адресом.

Шаг 5 Запустите сервер.

```
python manage.py runserver
```

Шаг 6

Просмотрите `http://localhost:8000/` в браузере, и вы увидите:

> Привет, мир

Виртуальная среда

Хотя это и не требуется строго, настоятельно рекомендуется начать свой проект в «виртуальной среде». Виртуальная среда - это **контейнер** (каталог), который содержит определенную версию Python и набор модулей (зависимостей) и который не мешает родному Python операционной системы или другим проектам на одном компьютере.

Путем настройки другой виртуальной среды для каждого проекта, над которым вы

работаете, различные проекты Django могут запускаться на разных версиях Python и могут поддерживать собственные наборы зависимостей без риска конфликтов.

Python 3.3+

Python 3.3+ уже включает стандартный модуль `venv` , который вы обычно можете назвать `pyvenv` . В средах, где команда `pyvenv` недоступна, вы можете получить доступ к тем же функциям, напрямую вызвав модуль как `python3 -m venv` .

Чтобы создать виртуальную среду:

```
$ pyvenv <env-folder>
# Or, if pyvenv is not available
$ python3 -m venv <env-folder>
```

Python 2

Если вы используете Python 2, вы можете сначала установить его как отдельный модуль из pip:

```
$ pip install virtualenv
```

Затем создайте среду, используя команду `virtualenv` :

```
$ virtualenv <env-folder>
```

Активировать (любая версия)

Теперь виртуальная среда настроена. Чтобы использовать его, он должен быть *активирован* в терминале, который вы хотите использовать.

Чтобы «активировать» виртуальную среду (любую версию Python)

Linux:

```
$ source <env-folder>/bin/activate
```

Windows:

```
<env-folder>\Scripts\activate.bat
```

Это изменяет ваше приглашение, чтобы указать, что виртуальная среда активна. `(<env-folder>) $`

Отныне все, установленное с помощью `pip` будет установлено в вашу виртуальную папку env, а не в масштабе всей системы.

Чтобы `deactivate` виртуальную среду, `deactivate` :

```
(<env-folder>) $ deactivate
```

Альтернативно: используйте virtualenvwrapper

Вы также можете рассмотреть возможность использования virtualenvwrapper, который делает создание и активацию virtualenv очень удобным, а также отделяет его от вашего кода:

```
# Create a virtualenv
mkvirtualenv my_virtualenv

# Activate a virtualenv
workon my_virtualenv

# Deactivate the current virtualenv
deactivate
```

Альтернативно: используйте pyenv + pyenv-viritualenv

В средах, где вам нужно обрабатывать несколько версий Python, вы можете использовать virtualenv вместе с pyenv-virtualenv:

```
# Create a virtualenv for specific Python version
pyenv virtualenv 2.7.10 my-virtual-env-2.7.10

# Create a vritualenv for active python verion
pyenv virtualenv venv34

# Activate, deactivate virtualenv
pyenv activate <name>
pyenv deactivate
```

При использовании virtualenvs часто бывает полезно установить `PYTHONPATH` и `DJANGO_SETTINGS_MODULE` в сценарии `postactivate` .

```
#!/bin/sh
# This hook is sourced after this virtualenv is activated
```

```
# Set PYTHONPATH to isolate the virtualenv so that only modules installed
# in the virtualenv are available
export PYTHONPATH="/home/me/path/to/your/project_root:$VIRTUAL_ENV/lib/python3.4"

# Set DJANGO_SETTINGS_MODULE if you don't use the default `myproject.settings`
# or if you use `django-admin` rather than `manage.py`
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Укажите путь к проекту

Часто также полезно указать путь к проекту в специальном файле `.project` расположенном в вашей базе `<env-folder>` . При этом каждый раз, когда вы активируете свою виртуальную среду, она будет изменять активный каталог на указанный путь.

Создайте новый файл с именем `<env-folder>/.project` . Содержимое файла должно быть ТОЛЬКО в качестве пути к каталогу проекта.

```
/path/to/project/directory
```

Теперь инициируйте свою виртуальную среду (используя `source <env-folder>/bin/activate` или `workon my_virtualenv`), и ваш терминал изменит каталоги в каталог `/path/to/project/directory` .

Пример простого файла Hello World

В этом примере вы покажете минимальный способ создания страницы Hello World в Django. Это поможет вам понять, что `django-admin startproject example` проекта `django-admin startproject example` основном создает кучу папок и файлов и что вам не обязательно нужна эта структура для запуска вашего проекта.

1. Создайте файл с именем `file.py`
2. Скопируйте и вставьте следующий код в этот файл.

```
import sys

from django.conf import settings

settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
```

```
from django.conf.urls import url
from django.http import HttpResponse

# Your code goes below this line.

def index(request):
    return HttpResponse('Hello, World!')

urlpatterns = [
    url(r'^$', index),
]

# Your code goes above this line

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

3. Перейдите к терминалу и запустите файл с помощью этой команды `python file.py runserver` .

4. Откройте браузер и перейдите к 127.0.0.1:8000 .

Дружественный к развертыванию проект с поддержкой Docker.

Шаблон проекта Django по умолчанию подходит, но как только вы доберетесь до развертывания своего кода, и, например, дефолты положили руки на проект, все становится беспорядочным. Что вы можете сделать, это отделить исходный код от остальных, которые должны быть в вашем репозитории.

> Вы можете найти подходящий шаблон проекта Django на GitHub .

Структура проекта

```
PROJECT_ROOT
├── devel.dockerfile
├── docker-compose.yml
├── nginx
│   └── project_name.conf
├── README.md
├── setup.py
└── src
    ├── manage.py
    └── project_name
        ├── __init__.py
        └── service
            ├── __init__.py
            ├── settings
            │   ├── common.py
            │   ├── development.py
            │   ├── __init__.py
            │   └── staging.py
```

```
        ├── urls.py
        └── wsgi.py
```

Мне нравится сохранять `service` каталог с именем `service` для каждого проекта, благодаря `Dockerfile` я могу использовать один и тот же `Dockerfile` во всех моих проектах. Разделение требований и настроек уже хорошо описано здесь:
Использование нескольких файлов требований
Использование нескольких настроек

Dockerfile

С предположением, что только разработчики используют Docker (не каждый dev ops доверяет ему в эти дни). Это может быть `devel.dockerfile` dev:

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1

RUN mkdir /run/service
ADD . /run/service
WORKDIR /run/service

RUN pip install -U pip
RUN pip install -I -e .[develop] --process-dependency-links

WORKDIR /run/service/src
ENTRYPOINT ["python", "manage.py"]
CMD ["runserver", "0.0.0.0:8000"]
```

При добавлении только требований будет использоваться кеш докеров при построении - вам нужно будет только перестроить при изменении требований.

компоновать

Docker compomp пригодится - особенно когда у вас есть несколько сервисов для локального запуска. `docker-compose.yml` :

```
version: '2'
services:
  web:
    build:
      context: .
      dockerfile: devel.dockerfile
    volumes:
      - "./src/{{ project_name }}:/run/service/src/{{ project_name }}"
      - "./media:/run/service/media"
    ports:
      - "8000:8000"
    depends_on:
      - db
```

```
  db:
    image: mysql:5.6
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE={{ project_name }}
  nginx:
    image: nginx
    ports:
      - "80:80"
    volumes:
      - "./nginx:/etc/nginx/conf.d"
      - "./media:/var/media"
    depends_on:
      - web
```

Nginx

Ваша среда разработки должна быть как можно ближе к среде prod, поэтому мне нравится использовать Nginx с самого начала. Вот пример файла конфигурации nginx:

```
server {
    listen   80;
    client_max_body_size 4G;
    keepalive_timeout 5;

    location /media/ {
        autoindex on;
        alias /var/media/;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Ssl on;
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_pass http://web:8000/;
    }
}
```

использование

```
$ cd PROJECT_ROOT
$ docker-compose build web  # build the image - first-time and after requirements change
$ docker-compose up  # to run the project
$ docker-compose run --rm --service-ports --no-deps  # to run the project - and be able to use
PDB
$ docker-compose run --rm --no-deps <management_command>  # to use other than runserver
commands, like makemigrations
```

```
$ docker exec -ti web bash  # For accessing django container shell, using it you will be
inside /run/service directory, where you can run ./manage shell, or other stuff
$ docker-compose start  # Starting docker containers
$ docker-compose stop  # Stopping docker containers
```

Прочитайте Начало работы с Django онлайн: https://riptutorial.com/ru/django/topic/200/начало-работы-с-django

глава 2: ArrayField - поле PostgreSQL

Синтаксис

- from django.contrib.postgres.fields импортировать ArrayField
- class ArrayField (base_field, size = None, ** options)
- FooModel.objects.filter (array_field_name__contains = [objects, to, check])
- FooModel.objects.filter (array_field_name__contained_by = [objects, to, check])

замечания

Обратите внимание, что хотя параметр `size` передается PostgreSQL, PostgreSQL не будет применять его.

При использовании `ArrayField` s следует помнить об этом предупреждении из документации по массивам Postgresql .

> Совет. Массивы не являются наборами; поиск определенных элементов массива может быть признаком неправильной настройки базы данных. Рассмотрите возможность использования отдельной таблицы со строкой для каждого элемента, который будет элементом массива. Это будет легче искать и, скорее всего, лучше масштабируется для большого количества элементов.

Examples

Основной массив ArrayField

Чтобы создать PostgreSQL ArrayField, мы должны предоставить ArrayField тип данных, которые мы хотим сохранить в качестве поля в качестве первого аргумента. Поскольку мы будем хранить рейтинги книг, мы будем использовать `FloatField` .

```
from django.db import models, FloatField
from django.contrib.postgres.fields import ArrayField

class Book(models.Model):
    ratings = ArrayField(FloatField())
```

Указание максимального размера массива ArrayField

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class IceCream(models.Model):
    scoops = ArrayField(IntegerField()  # we'll use numbers to ID the scoops
```

```
                    , size=6)  # our parlor only lets you have 6 scoops
```

Когда вы используете параметр размера, он передается postgresql, который принимает его, а затем игнорирует его! Таким образом, вполне возможно добавить 7 целых чисел в поле `scoops` выше, используя консоль postgresql.

Запрос на членство в ArrayField c

Этот запрос возвращает все конусы с шоколадным ковшом и ванильным ковшом.

```
VANILLA, CHOCOLATE, MINT, STRAWBERRY = 1, 2, 3, 4  # constants for flavors
choco_vanilla_cones = IceCream.objects.filter(scoops__contains=[CHOCOLATE, VANILLA])
```

Не забудьте импортировать `IceCream` модель из вашего `models.py` файла.

Также имейте в виду, что django не будет создавать индекс для `ArrayField` s. Если вы собираетесь их искать, вам понадобится индекс, и его нужно будет создать вручную с помощью вызова RunSQL в вашем файле миграции.

Вложенные массивы

Вы можете `ArrayField` s, передав другой `ArrayField` поскольку это `base_field` .

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class SudokuBoard(models.Model):
    numbers = ArrayField(
        ArrayField(
            models.IntegerField(),
            size=9,
        ),
        size=9,
    )
```

Запрос для всех моделей, которые содержат любой элемент в списке с contains_by

Этот запрос возвращает все конусы либо соломинкой, либо ванильным ковшом.

```
minty_vanilla_cones = IceCream.objects.filter(scoops__contained_by=[MINT, VANILLA])
```

Прочитайте ArrayField - поле PostgreSQL онлайн:
https://riptutorial.com/ru/django/topic/1693/arrayfield---поле-postgresql

глава 3: CRUD в Django

Examples

** Простейший пример CRUD **

Если вы обнаружите, что эти шаги незнакомы, подумайте о том, чтобы начать здесь . Обратите внимание, что эти шаги взяты из документации переполнения стека.

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

myproject / settings.py Установка приложения

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp',
]
```

Создайте файл `urls.py` в каталоге **myapp** и `urls.py` его следующим представлением.

```
from django.conf.urls import url
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Обновите файл `urls.py` другим содержимым.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^myapp/', include('myapp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Создайте папку с именем `templates` в каталоге **myapp** . Затем создайте файл с именем `index.html` внутри каталога **шаблонов** . Заполните его следующим содержанием.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

Нам также необходимо просмотреть **index.html,** который мы можем создать, отредактировав файл **views.py** следующим образом:

```
from django.shortcuts import render, redirect

# Create your views here.
def index(request):
    return render(request, 'index.html', {})
```

Теперь у вас есть база, от которой вы будете работать. Следующий шаг - создать модель. Это самый простой пример, поэтому в папке **models.py** добавьте следующий код.

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.
class Name(models.Model):
    name_value = models.CharField(max_length=100)

    def __str__(self): # if Python 2 use __unicode__
        return self.name_value
```

Это создает модель объекта Name, которую мы добавим в базу данных со следующими командами из командной строки.

```
python manage.py createsuperuser
python manage.py makemigrations
python manage.py migrate
```

Вы должны увидеть некоторые операции, выполненные Django. Они настраивают таблицы и создают суперпользователя, который может получить доступ к базе данных администратора из Django, представленного администратором. Говоря об этом, давайте зарегистрируем нашу новую модель с видом администратора. Перейдите к **admin.py** и добавьте следующий код.

```
from django.contrib import admin
from myapp.models import Name
# Register your models here.
```

```
admin.site.register(Name)
```

Вернувшись в командную строку, вы теперь можете `python manage.py runserver` сервер с помощью команды `python manage.py runserver` . Вы можете посетить http: // localhost: 8000 / и посмотреть свое приложение. Затем перейдите к http: // localhost: 8000 / admin, чтобы вы могли добавить имя в свой проект. Войдите в систему и добавьте имя в таблицу MYAPP, поэтому мы сохранили ее просто, чтобы убедиться, что она меньше 100 символов.

Чтобы получить доступ к названию, вам нужно отобразить его где-нибудь. Отредактируйте функцию index в **файле views.py,** чтобы получить все объекты Name из базы данных.

```
from django.shortcuts import render, redirect
from myapp.models import Name

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()
    context_dict = {'names_from_context': names_from_db}
    return render(request, 'index.html', context_dict)
```

Теперь отредактируйте файл **index.html** следующим образом.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

Это демонстрирует чтение в CRUD. В каталоге **myapp** создайте файл forms.py. Добавьте следующий код:

```
from django import forms
from myapp.models import Name

class NameForm(forms.ModelForm):
    name_value = forms.CharField(max_length=100, help_text = "Enter a name")
```

```
    class Meta:
        model = Name
        fields = ('name_value',)
```

Обновите **index.html** следующим образом:

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <form id="name_form" method="post" action="/">
        {% csrf_token %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}
            {{ field }}
        {% endfor %}
        <input type="submit" name="submit" value="Create">
    </form>
</body>
</html>
```

Затем обновите **view.py** следующим образом:

```
from django.shortcuts import render, redirect
from myapp.models import Name
from myapp.forms import NameForm

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()

    form = NameForm()

    context_dict = {'names_from_context': names_from_db, 'form': form}

    if request.method == 'POST':
        form = NameForm(request.POST)

        if form.is_valid():
            form.save(commit=True)
            return render(request, 'index.html', context_dict)
```

```
        else:
            print(form.errors)

    return render(request, 'index.html', context_dict)
```

Перезагрузите свой сервер, и теперь у вас должна быть рабочая версия приложения с созданным C в create.

TODO добавить обновление и удалить

Прочитайте CRUD в Django онлайн: https://riptutorial.com/ru/django/topic/7317/crud-в-django

глава 4: Django Rest Framework

Examples

Simple barebones API для чтения

Предполагая, что у вас есть модель, которая выглядит следующим образом, мы получим работу с простым API - интерфейсом **только для** браузера, основанным на Django REST Framework («DRF»).

models.py

```
class FeedItem(models.Model):
    title = models.CharField(max_length=100, blank=True)
    url = models.URLField(blank=True)
    style = models.CharField(max_length=100, blank=True)
    description = models.TextField(blank=True)
```

Сериализатор - это компонент, который будет извлекать всю информацию из модели Django (в данном случае `FeedItem`) и превращать ее в JSON. Он очень похож на создание классов форм в Django. Если у вас есть опыт в этом, вам будет очень удобно.

serializers.py

```
from rest_framework import serializers
from . import models

class FeedItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.FeedItem
        fields = ('title', 'url', 'description', 'style')
```

views.py

DRF предлагает множество классов просмотра для обработки различных вариантов использования. В этом примере мы будем иметь **только** API - интерфейс только для **чтения** , поэтому вместо использования более полного набора представлений или совокупности связанных общих представлений мы будем использовать один подкласс `ListAPIView` DRF.

Цель этого класса - связать данные с сериализатором и объединить все вместе для объекта ответа.

```
from rest_framework import generics
from . import serializers, models

class FeedItemList(generics.ListAPIView):
    serializer_class = serializers.FeedItemSerializer
```

```
    queryset = models.FeedItem.objects.all()
```

urls.py

Убедитесь, что вы указали маршрут на свой вид DRF.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    ...
    url(r'path/to/api', views.FeedItemList.as_view()),
]
```

Прочитайте Django Rest Framework онлайн: https://riptutorial.com/ru/django/topic/7341/django-rest-framework

глава 5: Django и социальные сети

параметры

настройка	Есть ли
Некоторые конфигурации	Удобные базовые настройки, которые идут с Django-Allauth (которые я использую большую часть времени). Дополнительные параметры конфигурации см. В разделе Конфигурации
ACCOUNT_AUTHENTICATION_METHOD (= «имя пользователя» или «электронная почта» или «имя_пользователя»)	Задает метод входа в систему - вводит ли пользователь вход, введя свое имя пользователя, адрес электронной почты или любой из них. Для этого для «email» требуется ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_CONFIRMATION_EXPIRE_DAYS (= 3)	Определяет дату истечения срока действия писем с подтверждением электронной почты (количество дней).
ACCOUNT_EMAIL_REQUIRED (= False)	Пользователь должен передать адрес электронной почты при регистрации. Это происходит в тандеме с настройкой `ACCOUNT_AUTHENTICATION_METHOD`
ACCOUNT_EMAIL_VERIFICATION (= " опционально")	Определяет метод проверки электронной почты во время регистрации - выберите один из «обязательный», «необязательный» или «none». Если установлено «обязательное», пользователь блокируется от входа в систему до тех пор, пока не будет проверен адрес электронной почты. Выберите «optional» или «none», чтобы разрешить логины с непроверенным адресом электронной почты. В случае «факультативного» почта электронной почты по-прежнему отправляется, тогда как в случае «ни

настройка	Есть ли
	одного» письма с проверкой электронной почты не отправляются.
ACCOUNT_LOGIN_ATTEMPTS_LIMIT (= 5)	Количество неудачных попыток входа в систему. Когда это число превышено, пользователю запрещается входить в систему для указанных ACCOUNT_LOGIN_ATTEMPTS_TIMEOUT секунд. Хотя это защищает просмотр входа allauth, он не защищает логин администратора Django от принудительного принуждения.
ACCOUNT_LOGOUT_ON_PASSWORD_CHANGE (= False)	Определяет, будет ли пользователь автоматически выходить из системы после изменения или установки своего пароля.
SOCIALACCOUNT_PROVIDERS (= dict)	Словарь, содержащий настройки конкретного поставщика.

Examples

Простой способ: python-social-auth

python-social-auth - это структура, упрощающая механизм социальной аутентификации и авторизации. Он содержит множество социальных бэкэндов (Facebook, Twitter, Github, LinkedIn и т. Д.)

УСТАНОВИТЬ

Сначала нам нужно установить пакет python-social-auth с

```
pip install python-social-auth
```

или загрузить код из github. Теперь самое время добавить это в файл `requirements.txt` .

НАСТРОЙКА settings.py

В settings.py добавьте:

```
INSTALLED_APPS = (
    ...
    'social.apps.django_app.default',
```

```
    ...
)
```

КОНФИГУРИРОВАНИЕ РЕЗЕРВОВ

AUTHENTICATION_BACKENDS содержит бэкэнды, которые мы будем использовать, и нам нужно только поставить то, что нам нужно.

```
AUTHENTICATION_BACKENDS = (
    'social.backends.open_id.OpenIdAuth',
    'social.backends.google.GoogleOpenId',
    'social.backends.google.GoogleOAuth2',
    'social.backends.google.GoogleOAuth',
    'social.backends.twitter.TwitterOAuth',
    'social.backends.yahoo.YahooOpenId',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Возможно, у вашего проекта `settings.py` еще нет поля `AUTHENTICATION_BACKENDS` . Если это так, добавьте это поле. Не забудьте пропустить `'django.contrib.auth.backends.ModelBackend',` поскольку он обрабатывает логин по имени пользователя / паролю.

Если мы используем, например, Facebook и Linkedin Backends, нам нужно добавить ключи API

```
SOCIAL_AUTH_FACEBOOK_KEY = 'YOURFACEBOOKKEY'
SOCIAL_AUTH_FACEBOOK_SECRET = 'YOURFACEBOOKSECRET'
```

а также

```
SOCIAL_AUTH_LINKEDIN_KEY = 'YOURLINKEDINKEY'
SOCIAL_AUTH_LINKEDIN_SECRET = 'YOURLINKEDINSECRET'
```

Примечание . Вы можете получить закрытые ключи в разработчиках Facebook и разработчиках Linkedin, и здесь вы можете увидеть полный список и его соответствующий способ узнать ключ API и ключ Secret.

Примечание по секретным ключам: секретные ключи должны храниться в секрете. Ниже приведено описание переполнения стека, которое полезно. Этот учебник полезен для изучения переменных среды.

TEMPLATE_CONTEXT_PROCESSORS помогут переадресации, backend и другие вещи, но в начале нам нужны только эти:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social.apps.django_app.context_processors.backends',
    'social.apps.django_app.context_processors.login_redirect',
    ...
```

```
)
```

В Django 1.8 настройка `TEMPLATE_CONTEXT_PREPROCESSORS` как показано выше, устарела. Если это так, вы добавите его внутри `TEMPLATES` dict. Ваш взгляд должен выглядеть примерно так:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social.apps.django_app.context_processors.backends',
                'social.apps.django_app.context_processors.login_redirect',
            ],
        },
    },
]
```

ИСПОЛЬЗОВАНИЕ ТАМОЖЕННОГО ПОЛЬЗОВАТЕЛЯ

Если вы используете пользовательскую модель пользователя и хотите присоединиться к ней, просто добавьте следующую строку (все еще в **settings.py**)

```
SOCIAL_AUTH_USER_MODEL = 'somepackage.models.CustomUser'
```

`CustomUser` - это модель, наследующая или абстрактная от пользователя по умолчанию.

КОНФИГУРАЦИЯ urls.py

```
# if you haven't imported inlcude make sure you do so at the top of your file
from django.conf.urls import url, include

urlpatterns = patterns('',
    ...
    url('', include('social.apps.django_app.urls', namespace='social'))
    ...
)
```

Затем необходимо синхронизировать базу данных для создания необходимых моделей:

```
./manage.py migrate
```

Наконец мы можем играть!

в каком-то шаблоне вам нужно добавить что-то вроде этого:

```
    <a href="{% url 'social:begin' 'facebook' %}?next={{ request.path }}">Login with
```

```
Facebook</a>
    <a href="{% url 'social:begin' 'linkedin' %}?next={{ request.path }}">Login with
Linkedin</a>
```

если вы используете другой бэкэнд, просто измените «facebook» на имя бэкэнда.

Запуск пользователей

После того, как вы зарегистрировали пользователей, вы, скорее всего, захотите создать функциональность для их регистрации. В некоторых шаблонах, вероятно, рядом с тем, где был показан шаблон журнала, добавьте следующий тег:

```
<a href="{% url 'logout' %}">Logout</a>
```

или же

```
<a href="/logout">Logout</a>
```

Вы захотите отредактировать файл `urls.py` с кодом, похожим на:

```
url(r'^logout/$', views.logout, name='logout'),
```

Наконец, отредактируйте файл views.py с кодом, похожим на:

```
def logout(request):
    auth_logout(request)
    return redirect('/')
```

Использование Django Allauth

Для всех моих проектов Django-Allauth оставался тем, который легко настраивается, и выходит из коробки со многими функциями, включая, но не ограничиваясь:

- Примерно 50+ аутентификации социальных сетей
- Смешать регистрацию как локальных, так и социальных учетных записей
- Несколько социальных счетов
- Необязательная мгновенная регистрация для социальных учетных записей - без вопросов
- Управление адресами электронной почты (несколько адресов электронной почты, настройка первичного)
- Потерянный поток потока поток проверки адреса электронной почты

Если вы заинтересованы в загрязнении рук, Django-Allauth убирается с пути, с дополнительными настройками, чтобы настроить процесс и использовать вашу систему аутентификации.

Следующие шаги предполагают, что вы используете Django 1.10+

Шаги настройки:

```
pip install django-allauth
```

В файле `settings.py` внесите следующие изменения:

```
# Specify the context processors as follows:
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # Already defined Django-related contexts here

                # `allauth` needs this from django. It is there by default,
                # unless you've devilishly taken it away.
                'django.template.context_processors.request',
            ],
        },
    },
]

AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    'django.contrib.auth.backends.ModelBackend',

    # `allauth` specific authentication methods, such as login by e-mail
    'allauth.account.auth_backends.AuthenticationBackend',
)

INSTALLED_APPS = (
# Up here is all your default installed apps from Django

# The following apps are required:
'django.contrib.auth',
'django.contrib.sites',

'allauth',
'allauth.account',
'allauth.socialaccount',

# include the providers you want to enable:
'allauth.socialaccount.providers.google',
'allauth.socialaccount.providers.facebook',
)

# Don't forget this little dude.
SITE_ID = 1
```

`urls.py` изменения в файле `settings.py` выше, перейдите в файл `urls.py` Это может быть ваш `yourapp/urls.py` или ваш `ProjectName/urls.py` Обычно я предпочитаю `ProjectName/urls.py`

```
urlpatterns = [
    # other urls here
```

```
    url(r'^accounts/', include('allauth.urls')),
    # other urls here
]
```

Просто добавив `include('allauth.urls')` , вы получите эти URL бесплатно:

```
^accounts/ ^ ^signup/$ [name='account_signup']
^accounts/ ^ ^login/$ [name='account_login']
^accounts/ ^ ^logout/$ [name='account_logout']
^accounts/ ^ ^password/change/$ [name='account_change_password']
^accounts/ ^ ^password/set/$ [name='account_set_password']
^accounts/ ^ ^inactive/$ [name='account_inactive']
^accounts/ ^ ^email/$ [name='account_email']
^accounts/ ^ ^confirm-email/$ [name='account_email_verification_sent']
^accounts/ ^ ^confirm-email/(?P<key>[-:\w]+)/$ [name='account_confirm_email']
^accounts/ ^ ^password/reset/$ [name='account_reset_password']
^accounts/ ^ ^password/reset/done/$ [name='account_reset_password_done']
^accounts/ ^ ^password/reset/key/(?P<uidb36>[0-9A-Za-z]+)-(?P<key>.+)/$
[name='account_reset_password_from_key']
^accounts/ ^ ^password/reset/key/done/$ [name='account_reset_password_from_key_done']
^accounts/ ^social/
^accounts/ ^google/
^accounts/ ^twitter/
^accounts/ ^facebook/
^accounts/ ^facebook/login/token/$ [name='facebook_login_by_token']
```

Наконец, выполните `python ./manage.py migrate` чтобы `python ./manage.py migrate` Django-allauth в базу данных.

Как обычно, чтобы иметь возможность войти в ваше приложение с помощью какой-либо социальной сети, которую вы добавили, вам нужно будет добавить информацию о социальной учетной записи в сети.

Войдите в администратор Django (`localhost:8000/admin`) и в разделе « `Social Applications` чтобы добавить информацию о своей социальной учетной записи.

Для получения подробной информации о заполнении в разделах «Социальные приложения» вам могут потребоваться учетные записи у каждого поставщика авторизации.

Подробные сведения о том, что вы можете иметь и настроить, см. На странице «Конфигурации» .

Прочитайте Django и социальные сети онлайн:
https://riptutorial.com/ru/django/topic/4743/django-и-социальные-сети

глава 6: Django из командной строки.

замечания

В то время как Django предназначен в основном для веб-приложений, он имеет мощную и удобную в использовании ORM, которая также может использоваться для приложений командной строки и скриптов. Существует два разных подхода. Первым из них является создание пользовательской команды управления, а вторая - инициализация среды Django в начале вашего скрипта.

Examples

Django из командной строки.

Предположим, что вы настроили проект django, а файл настроек находится в приложении с именем main, так вы инициализируете свой код

```
import os, sys

# Setup environ
sys.path.append(os.getcwd())
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "main.settings")

# Setup django
import django
django.setup()

# rest of your imports go here

from main.models import MyModel

# normal python code that makes use of Django models go here

for obj in MyModel.objects.all():
    print obj
```

Вышеприведенное может быть выполнено как

```
python main/cli.py
```

Прочитайте Django из командной строки. онлайн:
https://riptutorial.com/ru/django/topic/5848/django-из-командной-строки-

глава 7: FormSets

Синтаксис

- NewFormSet = formset_factory (SomeForm, extra = 2)
- formet = NewFormSet (initial = [{'some_field': 'Field Value', 'other_field': 'Other Field Value',}])

Examples

Формы с инициализированными и унифицированными данными

`Formset` - способ отображения нескольких форм на одной странице, например, сетки данных. Пример: этот `ChoiceForm` может быть связан с некоторым вопросом рода. как, дети самые умные, между которыми возраст ?.

`appname/forms.py`

```
from django import forms
class ChoiceForm(forms.Form):
    choice = forms.CharField()
    pub_date = forms.DateField()
```

В ваших представлениях вы можете использовать конструктор `formset_factory` который принимает `Form` в качестве параметра `ChoiceForm` в этом случае и `extra` который описывает, сколько дополнительных форм, кроме инициализированных форм / форм, нужно визуализировать, и вы можете `formset` объект `formset` же, как любой другой итерируемый.

Если формат не инициализирован данными, он печатает число форм, равное `extra + 1` и если инициализируется набор форм, он печатает `initialized + extra` где `extra` количество пустых форм, отличных от инициализированных.

`appname/views.py`

```
import datetime
from django.forms import formset_factory
from appname.forms import ChoiceForm
    ChoiceFormSet = formset_factory(ChoiceForm, extra=2)
    formset = ChoiceFormSet(initial=[
      {'choice': 'Between 5-15 ?',
        'pub_date': datetime.date.today(),}
      ])
```

если вы `formset object` подобный этому для формы в formet: print (form.as_table ())

`Output in rendered template`

```
<tr>
<th><label for="id_form-0-choice">Choice:</label></th>
<td><input type="text" name="form-0-choice" value="Between 5-15 ?" id="id_form-0-choice"
/></td>
</tr>
<tr>
<th><label for="id_form-0-pub_date">Pub date:</label></th>
<td><input type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date"
/></td>
</tr>
<tr>
<th><label for="id_form-1-choice">Choice:</label></th>
<td><input type="text" name="form-1-choice" id="id_form-1-choice" /></td>
</tr>
<tr>
<th><label for="id_form-1-pub_date">Pub date:</label></th>
<td><input type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td
</tr>
<tr>
<th><label for="id_form-2-choice">Choice:</label></th>
<td><input type="text" name="form-2-choice" id="id_form-2-choice" /></td>
</tr>
<tr>
<th><label for="id_form-2-pub_date">Pub date:</label></th>
<td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td>
</tr>
```

Прочитайте FormSets онлайн: https://riptutorial.com/ru/django/topic/6082/formsets

глава 8: JSONField - поле PostgreSQL

Синтаксис

- JSONField (** опция)

замечания

- `JSONField` Django фактически хранит данные в столбце Postgres `JSONB` , который доступен только в Postgres 9.4 и более поздних версиях.
- `JSONField` подходит для более гибкой схемы. Например, если вы хотите изменить ключи, не выполняя никаких миграций данных, или если не все ваши объекты имеют одинаковую структуру.
- Если вы храните данные со статическими ключами, вместо этого вместо использования `JSONField` использовать несколько нормальных полей, так как запрос `JSONField` может быть довольно утомительным.

Цепочки запросов

Вы можете связать запросы вместе. Например, если словарь существует внутри списка, добавьте два символа подчеркивания и ваш словарь.

> Не забудьте разделить запросы с двойными подчеркиваниями.

Examples

Создание JSONField

Доступно в Django 1.9+

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class IceCream(models.Model):
    metadata = JSONField()
```

Вы можете добавить обычные `**options` если хотите.

> ***!*** Обратите внимание, что вы должны поместить `'django.contrib.postgres'` в `INSTALLED_APPS` в `settings.py`

Создание объекта с данными в JSONField

Передавать данные в родной форме Python, например `list` , `dict` , `str` , `None` , `bool` и т. Д.

```
IceCream.objects.create(metadata={
    'date': '1/1/2016',
    'ordered by': 'Jon Skeet',
    'buyer': {
         'favorite flavor': 'vanilla',
         'known for': ['his rep on SO', 'writing a book']
     },
    'special requests': ['hot sauce'],
})
```

См. Примечание в разделе «Замечания» об использовании `JSONField` на практике.

Запрос данных верхнего уровня

```
IceCream.objects.filter(metadata__ordered_by='Guido Van Rossum')
```

Запрос данных, вложенных в словари

Получите все конусы мороженого, которые были заказаны людьми, любящими шоколад:

```
IceCream.objects.filter(metadata__buyer__favorite_flavor='chocolate')
```

См. Примечание в разделе «Примечания» о цепочке запросов.

Запрос данных, присутствующих в массивах

Целое число будет интерпретироваться как индексный поиск.

```
IceCream.objects.filter(metadata__buyer__known_for__0='creating stack overflow')
```

См. Примечание в разделе «Примечания» о цепочке запросов.

Заказ по значениям JSONField

Заказ на `JSONField` еще не поддерживается в Django. Но это возможно через RawSQL, используя функции PostgreSQL для jsonb:

```
from django.db.models.expressions import RawSQL
RatebookDataEntry.objects.all().order_by(RawSQL("data->>%s", ("json_objects_key",)))
```

Этот пример упорядочивает `data['json_objects_key']` внутри `JSONField` именем `data` :

```
data = JSONField()
```

Прочитайте JSONField - поле PostgreSQL онлайн:
https://riptutorial.com/ru/django/topic/1759/jsonfield---поле-postgresql

глава 9: Meta: Руководство по документации

замечания

Это расширение Python's Meta: Documentation Guidelines для Django.

Это всего лишь предложения, а не рекомендации. Не стесняйтесь редактировать что-либо здесь, если вы не согласны или имеете что-то еще, чтобы упомянуть.

Examples

Неподдерживаемые версии не требуют особого упоминания

Маловероятно, что кто-то использует неподдерживаемую версию Django и на свои собственные риски. Если кто-то делает это, его должно знать, существует ли функция в данной версии.

Учитывая вышеизложенное, бесполезно упоминать особенности неподдерживаемой версии.

1,6

Такой блок бесполезен, потому что ни один здравомыслящий человек не использует Django <1.6.

1,8

Такой блок бесполезен, потому что ни один здравомыслящий человек не использует Django <1.8.

Это также касается тем. На момент написания этого примера в представлениях, основанных на классе, поддерживаются версии `1.3-1.9` . Мы можем с уверенностью предположить, что это фактически эквивалентно `All versions` . Это также позволяет избежать обновления всех поддерживаемых версий тем каждый раз, когда выпущена новая версия.

> Поддерживаемые версии: `1.8` [1] `1.9` [2] `1.10` [1]

1. Исправления безопасности, ошибки потери данных, сбои в ошибках, ошибки основной функциональности в новых функциях и регрессии из более ранних версий Django.
2. Исправления безопасности и ошибки потери данных.

Прочитайте Meta: Руководство по документации онлайн:
https://riptutorial.com/ru/django/topic/5243/meta--руководство-по-документации

глава 10: Querysets

Вступление

`Queryset` - это в основном список объектов, полученных из `Model` , путем компиляции запросов к базе данных.

Examples

Простые запросы на автономную модель

Вот простая модель, которую мы будем использовать для запуска нескольких тестовых запросов:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Получить один объект модели, где id / pk равно 4:
(Если нет элементов с идентификатором 4 или их более одного, это вызовет исключение).

```
MyModel.objects.get(pk=4)
```

Все объекты модели:

```
MyModel.objects.all()
```

Объекты модели, для которых установлен `flag` `True` :

```
MyModel.objects.filter(flag=True)
```

Объекты модели с `model_num` больше 25:

```
MyModel.objects.filter(model_num__gt=25)
```

Объекты модели с `name` «Дешевый элемент» и `flag` установленный в значение « `False` :

```
MyModel.objects.filter(name="Cheap Item", flag=False)
```

Моделирует простое `name` поиска для конкретной строки (с учетом регистра):

```
MyModel.objects.filter(name__contains="ch")
```

Модели простое `name` поиска для конкретной строки (нечувствительность к регистру):

```
MyModel.objects.filter(name__icontains="ch")
```

Расширенные запросы с объектами Q

Учитывая модель:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Мы можем использовать объекты `Q` для создания условий `AND` , `OR` в вашем поисковом запросе. Например, предположим, что мы хотим, чтобы все объекты имели `flag=True` **OR** `model_num>15` .

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15))
```

Вышеприведенное `WHERE flag=True OR model_num > 15` аналогично для **AND, которое** вы бы сделали.

```
MyModel.objects.filter(Q(flag=True) & Q(model_num__gt=15))
```

Объекты `Q` также позволяют нам делать **HE-** запросы с использованием `~` . Предположим, мы хотели получить все объекты с `flag=False` **AND** `model_num!=15` , мы бы сделали:

```
MyModel.objects.filter(Q(flag=True) & ~Q(model_num=15))
```

При использовании объектов Q и «нормальных» параметров в `filter()` , объекты `Q` должны быть *первыми* . Следующий запрос выполняет поиск моделей с (`flag` установлен на `True` или номер модели больше `15`) и имя, которое начинается с «H».

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15), name__startswith="H")
```

Примечание. Объекты `Q` могут использоваться с любой функцией поиска, которая принимает аргументы ключевого слова, такие как `filter` , `exclude` , `get` . Убедитесь в том , что при использовании с `get` , что вы будете возвращать только один объект или `MultipleObjectsReturned` будет сгенерировано исключение.

Уменьшите количество запросов на ManyToManyField (проблема n + 1)

проблема

```
# models.py:
class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)

class Book(models.Model):
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.all()

    # Query the database on each iteration (len(author) times)
    # if there is 100 librairies, there will have 100 queries plus the initial query
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 101 queries
```

Решение

Используйте `prefetch_related` на `ManyToManyField` если вы знаете, что вам нужно будет получить доступ к полю, которое является полем `ManyToManyField` .

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()

    # Does not query the database again, since `books` is pre-populated
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 2 queries - 1 for libraries, 1 for books
```

`prefetch_related` также может использоваться для полей поиска:

```
# models.py:
class User(models.Model):
    name = models.CharField(max_length=100)

class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    readers = models.ManyToManyField(User)
```

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books', 'books__readers').all()

    # Does not query the database again, since `books` and `readers` is pre-populated
    for library in libraries:
        for book in library.books.all():
            for user in book.readers.all():
                user.name
                # ...

    # total : 3 queries - 1 for libraries, 1 for books, 1 for readers
```

Однако, как только запрос будет выполнен, данные не могут быть изменены без повторного использования базы данных. Следующие могут выполнять дополнительные запросы, например:

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()
    for library in libraries:
        for book in library.books.filter(title__contains="Django"):
            print(book.name)
```

Следующие могут быть оптимизированы с использованием объекта `Prefetch` , введенного в Django 1.7:

```
from django.db.models import Prefetch
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related(
        Prefetch('books', queryset=Book.objects.filter(title__contains="Django")
    ).all()
    for library in libraries:
        for book in library.books.all():
            print(book.name)  # Will print only books containing Django for each library
```

Уменьшить количество запросов в поле ForeignKey (n + 1 выпуск)

проблема

Запросы Django оцениваются ленивым способом. Например:

```
# models.py:
```

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database
    books = Book.objects.all()

    for book in books:
        # Query the database on each iteration to get author (len(books) times)
        # if there is 100 books, there will have 100 queries plus the initial query
        book.author
        # ...

    # total : 101 queries
```

Приведенный выше код заставляет django запрашивать базу данных для автора каждой книги. Это неэффективно, и лучше иметь только один запрос.

Решение

Используйте `select_related` on `ForeignKey` если вы знаете, что вам нужно будет позже получить доступ к полю `ForeignKey` .

```
# views.py
def myview(request):
    # Query the database.
    books = Books.objects.select_related('author').all()

    for book in books:
        # Does not query the database again, since `author` is pre-populated
        book.author
        # ...

    # total : 1 query
```

`select_related` также может использоваться для полей поиска:

```
# models.py:
class AuthorProfile(models.Model):
    city = models.CharField(max_length=100)

class Author(models.Model):
    name = models.CharField(max_length=100)
    profile = models.OneToOneField(AuthorProfile)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    books = Book.objects.select_related('author')\
                        .select_related('author__profile').all()

    for book in books:
        # Does not query database
        book.author.name
        # or
        book.author.profile.city
        # ...

    # total : 1 query
```

Получить SQL для набора запросов Django

Атрибут `query` дает синтаксис SQL-эквивалента для вашего запроса.

```
>>> queryset = MyModel.objects.all()
>>> print(queryset.query)
SELECT "myapp_mymodel"."id", ... FROM "myapp_mymodel"
```

> **Предупреждение:**
>
> Этот вывод следует использовать только для целей отладки. Сгенерированный запрос не зависит от сервера. Таким образом, параметры не котируются должным образом, что делает его уязвимым для SQL-инъекции, и запрос может даже не выполняться в бэкэнд базы данных.

Получить первую и последнюю запись из QuerySet

Чтобы получить первый объект:

```
MyModel.objects.first()
```

Чтобы получить последние объекты:

```
MyModel.objects.last()
```

Использование объекта Filter First:

```
MyModel.objects.filter(name='simple').first()
```

Использование фильтра Последний объект:

```
MyModel.objects.filter(name='simple').last()
```

Расширенные запросы с объектами F

> Объект F () представляет значение поля модели или аннотированного столбца. Это позволяет ссылаться на значения полей модели и выполнять операции с ними, не выбирая их из базы данных в память Python. - выражения F ()

Уместно использовать объекты `F()` когда вам нужно ссылаться на значение другого поля в вашем запросе. Само по себе объекты `F()` ничего не значат, и их нельзя и не следует вызывать вне набора запросов. Они используются для ссылки на значение поля в том же наборе запросов.

Например, учитывая модель ...

```
SomeModel(models.Model):
    ...
    some_field = models.IntegerField()
```

... пользователь может запрашивать объекты, где значение `some_field` в два раза превышает его `id` , ссылаясь на значение поля `id` при фильтрации с помощью `F()` следующим образом:

```
SomeModel.objects.filter(some_field=F('id') * 2)
```

`F('id')` просто ссылается на значение `id` для этого же экземпляра. Django использует его для создания соответствующего оператора SQL. В этом случае что-то очень похожее на это:

```
SELECT * FROM some_app_some_model
WHERE some_field = ((id * 2))
```

Без выражений `F()` это было бы достигнуто либо с помощью исходного SQL, либо с помощью фильтрации в Python (что снижает производительность, особенно когда есть много объектов).

Рекомендации:

- Фильтры могут ссылаться на поля на модели
- Выражения F
- Ответ от TinyInstance

Из определения класса `F()` :

> Объект, способный разрешать ссылки на существующие объекты запросов. - источник F

Примечание. Этот пример отправлен из ответа, указанного выше, с согласия TinyInstance.

Прочитайте Querysets онлайн: https://riptutorial.com/ru/django/topic/1235/querysets

глава 11: RangeFields - группа полей PostgreSQL

Синтаксис

- from django.contrib.postgres.fields import * RangeField
- IntegerRangeField (** опция)
- BigIntegerRangeField (** опция)
- FloatRangeField (** опция)
- DateTimeRangeField (** опция)
- DateRangeField (** опция)

Examples

Включая поля числовой области в вашей модели

В Python существует три типа числового `RangeField` s. `IntegerField` , `BigIntegerField` и `FloatField` . Они преобразуются в `psycopg2` `NumericRange` s, но принимают входные данные как собственные кортежи Python. **Нижняя граница включена и верхняя граница исключена.**

```
class Book(models.Model):
    name = CharField(max_length=200)
    ratings_range = IntegerRange()
```

Настройка для RangeField

1. добавьте `'django.contrib.postgres'` к вашему `INSTALLED_APPS`
2. установить `psycopg2`

Создание моделей с полями числового диапазона

Проще и проще вводить значения в виде кортежа Python вместо `NumericRange` .

```
Book.objects.create(name='Pro Git', ratings_range=(5, 5))
```

Альтернативный метод с `NumericRange` :

```
Book.objects.create(name='Pro Git', ratings_range=NumericRange(5, 5))
```

Использование содержит

Этот запрос выбирает все книги с любой оценкой менее трех.

```
bad_books = Books.objects.filter(ratings_range__contains=(1, 3))
```

Использование contains_by

Этот запрос получает все книги с рейтингами, большими или равными нулю и менее шести.

```
all_books = Book.objects.filter(ratings_range_contained_by=(0, 6))
```

Использование перекрытия

Этот запрос получает все перекрывающиеся встречи от шести до десяти.

```
Appointment.objects.filter(time_span__overlap=(6, 10))
```

Использование None для обозначения верхней границы

Этот запрос выбирает все книги с любым рейтингом, большим или равным четырем.

```
maybe_good_books = Books.objects.filter(ratings_range__contains=(4, None))
```

Операции с диапазонами

```
from datetime import timedelta

from django.utils import timezone
from psycopg2.extras import DateTimeTZRange

# To create a "period" object we will use psycopg2's DateTimeTZRange
# which takes the two datetime bounds as arguments
period_start = timezone.now()
period_end = period_start + timedelta(days=1, hours=3)
period = DateTimeTZRange(start, end)

# Say Event.timeslot is a DateTimeRangeField

# Events which cover at least the whole selected period,
Event.objects.filter(timeslot__contains=period)

# Events which start and end within selected period,
Event.objects.filter(timeslot__contained_by=period)

# Events which, at least partially, take place during the selected period.
Event.objects.filter(timeslot__overlap=period)
```

Прочитайте RangeFields - группа полей PostgreSQL онлайн:
https://riptutorial.com/ru/django/topic/2630/rangefields---группа-полей-postgresql

глава 12: администрация

Examples

Список изменений

Допустим, у вас есть простое приложение `myblog` со следующей моделью:

```
from django.conf import settings
from django.utils import timezone

class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70, unique=True)
    author = models.ForeignKey(settings.AUTH_USER_MODEL, models.PROTECT)
    date_published = models.DateTimeField(default=timezone.now)
    is_draft = models.BooleanField(default=True)
    content = models.TextField()
```

Django Admin «Список изменений» - это страница, в которой перечислены все объекты данной модели.

```
from django.contrib import admin
from myblog.models import Article

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    pass
```

По умолчанию он будет использовать метод `__str__()` (или `__unicode__()` если вы на python2) вашей модели, чтобы отобразить «имя» объекта. Это означает, что если вы не отменили его, вы увидите список статей, все с именем «Объект статьи». Чтобы изменить это поведение, вы можете установить метод `__str__()` :

```
class Article(models.Model):
    def __str__(self):
        return self.title
```

Теперь все ваши статьи должны иметь другое имя и более четко, чем «Объект статьи».

Однако вы можете отображать другие данные в этом списке. Для этого используйте `list_display` :

```
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['__str__', 'author', 'date_published', 'is_draft']
```

`list_display` не ограничивается полями и свойствами модели. он также может быть

методом вашего `ModelAdmin` :

```
from django.forms.utils import flatatt
from django.urls import reverse
from django.utils.html import format_html

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'author_link', 'date_published', 'is_draft']

    def author_link(self, obj):
        author = obj.author
        opts = author._meta
        route = '{}_{}_change'.format(opts.app_label, opts.model_name)
        author_edit_url = reverse(route, args=[author.pk])
        return format_html(
            '<a{}>{}</a>', flatatt({'href': author_edit_url}), author.first_name)

    # Set the column name in the change list
    author_link.short_description = "Author"
    # Set the field to use when ordering using this column
    author_link.admin_order_field = 'author__firstname'
```

Дополнительные стили CSS и сценарии JS для страницы администратора

Предположим, у вас простая модель `Customer` :

```
class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    is_premium = models.BooleanField(default=False)
```

Вы регистрируете его в администраторе Django и добавляете поле поиска по имени `first_name` и `last_name` :

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']
```

После этого поля поиска появятся на странице списка администраторов с пометкой по умолчанию: « *keyword* ». Но что, если вы хотите изменить этот заполнитель на « *Поиск по имени* »?

Вы можете сделать это, отправив пользовательский файл Javascript в admin `Media` :

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']

    class Media:
        #this path may be any you want,
```

```
        #just put it in your static folder
        js = ('js/admin/placeholder.js', )
```

Вы можете использовать панель инструментов debug debug, чтобы определить, какой идентификатор или класс Django установлен для этой панели поиска, а затем написать код js:

```
$(function () {
   $('#searchbar').attr('placeholder', 'Search by name')
})
```

Также `Media` класс позволяет добавлять файлы css со словарным объектом:

```
class Media:
    css = {
        'all': ('css/admin/styles.css',)
         }
```

Например, нам нужно отобразить каждый элемент столбца `first_name` в определенном цвете.
По умолчанию Django создает столбец таблицы для каждого элемента в `list_display` , все теги `<td>` будут иметь класс css, такой как `field-'list_display_name'` , в нашем случае он будет `field_first_name`

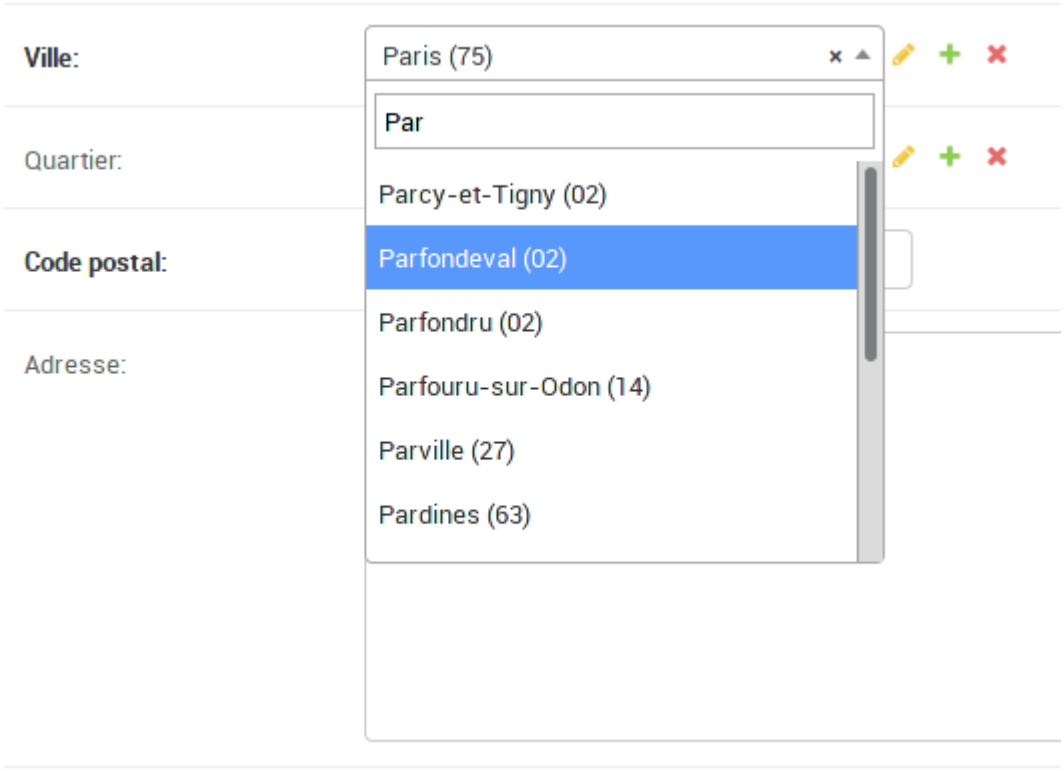
```
.field_first_name {
     background-color: #e6f2ff;
 }
```

Если вы хотите настроить другое поведение, добавив JS или некоторые стили CSS, вы всегда можете проверить id `s и классы элементов в инструменте отладки браузера.

Работа с внешними ключами, ссылающимися на большие таблицы

По умолчанию Django отображает поля `ForeignKey` как `<select>` . Это может привести к тому, что страницы будут **загружаться очень медленно,** если у вас есть тысячи или десятки тысяч записей в ссылочной таблице. И даже если у вас есть только сотни записей, довольно неудобно искать определенную запись среди всех.

Для этого очень удобный внешний модуль - django-autocomplete-light (DAL). Это позволяет использовать поля автозаполнения вместо полей `<select>` .



views.py

```
from dal import autocomplete

class CityAutocomp(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        qs = City.objects.all()
        if self.q:
            qs = qs.filter(name__istartswith=self.q)
        return qs
```

urls.py

```
urlpatterns = [
    url(r'^city-autocomp/$', CityAutocomp.as_view(), name='city-autocomp'),
]
```

forms.py

```
from dal import autocomplete

class PlaceForm(forms.ModelForm):
    city = forms.ModelChoiceField(
        queryset=City.objects.all(),
        widget=autocomplete.ModelSelect2(url='city-autocomp')
    )

    class Meta:
        model = Place
```

```
        fields = ['__all__']
```

admin.py

```
@admin.register(Place)
class PlaceAdmin(admin.ModelAdmin):
    form = PlaceForm
```

Прочитайте администрация онлайн: https://riptutorial.com/ru/django/topic/1219/администрация

глава 13: Асинхронные задачи (сельдерей)

замечания

Сельдерей - это очередь задач, которая может запускать фоновые или запланированные задания и очень хорошо интегрируется с Django. Сельдерей требует, чтобы что-то известное как **брокер сообщений** передавало сообщения от призыва к работникам. Этот брокер сообщений может быть redis, rabbitmq или даже Django ORM / db, хотя это не рекомендуется.

Прежде чем начать с примера, вам нужно будет настроить сельдерей. Чтобы настроить сельдерей, создайте файл `celery_config.py` в главном приложении, параллельно файлу `settings.py` .

```
from __future__ import absolute_import
import os
from celery import Celery
from django.conf import settings

# broker url
BROKER_URL = 'redis://localhost:6379/0'

# Indicate Celery to use the default Django settings module
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('config')
app.config_from_object('django.conf:settings')
# if you do not need to keep track of results, this can be turned off
app.conf.update(
    CELERY_RESULT_BACKEND=BROKER_URL,
)

# This line will tell Celery to autodiscover all your tasks.py that are in your app folders
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

И в файле `__init__.py` основного приложения импортируйте приложение celery. как это

```
# -*- coding: utf-8 -*-
# Not required for Python 3.
from __future__ import absolute_import

from .celery_config import app as celery_app  # noqa
```

Чтобы запустить работника сельдерея, используйте эту команду на уровне, где находится manage.py.

```
# pros is your django project,
celery -A proj worker -l info
```

Examples

Простой пример добавления двух чисел

Для начала:

1. Установить сельдерей `pip install celery`
2. настроить сельдерей (перейти в раздел замечаний)

```
from __future__ import absolute_import, unicode_literals

from celery.decorators import task


@task
def add_number(x, y):
    return x + y
```

Вы можете запустить это асинхронно с помощью `.delay()` .

`add_number.delay(5, 10)` , где 5 и 10 являются аргументами для функции `add_number`

Чтобы проверить, завершила ли функция async операцию, вы можете использовать функцию `.ready()` для объекта async, возвращаемого методом `delay` .

Чтобы получить результат вычисления, вы можете использовать атрибут `.result` для объекта async.

пример

```
async_result_object = add_number.delay(5, 10)
if async_result_object.ready():
    print(async_result_object.result)
```

Прочитайте Асинхронные задачи (сельдерей) онлайн:
https://riptutorial.com/ru/django/topic/5481/асинхронные-задачи--сельдерей-

глава 14: Аутентификация

Examples

Проверка подлинности электронной почты

Проверка подлинности по умолчанию Django работает с полями `username` и `password` . Брандмауэр аутентификации электронной почты будет аутентифицировать пользователей на основе `email` и `password` .

```
from django.contrib.auth import get_user_model

class EmailBackend(object):
    """
    Custom Email Backend to perform authentication via email
    """
    def authenticate(self, username=None, password=None):
        user_model = get_user_model()
        try:
            user = user_model.objects.get(email=username)
            if user.check_password(password): # check valid password
                return user # return user to be authenticated
        except user_model.DoesNotExist: # no matching user exists
            return None

    def get_user(self, user_id):
        user_model = get_user_model()
        try:
            return user_model.objects.get(pk=user_id)
        except user_model.DoesNotExist:
            return None
```

Добавьте этот аутентификационный сервер в настройку `AUTHENTICATION_BACKENDS` .

```
# settings.py
AUTHENTICATION_BACKENDS = (
    'my_app.backends.EmailBackend',
    ...
    )
```

Прочитайте Аутентификация онлайн: https://riptutorial.com/ru/django/topic/1282/аутентификация

глава 15: Безопасность

Examples

Защита от перекрестных ссылок (XSS)

Атаки XSS состоят в том, чтобы вводить HTML (или JS) код на странице. Дополнительные сведения см. В разделе « Межсайтовый скриптинг» .

Чтобы предотвратить эту атаку, по умолчанию Django избегает строк, прошедших через переменную шаблона.

Учитывая следующий контекст:

```
context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': (
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&quot; style=&quot;font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

Если у вас есть переменные, содержащие HTML, которым вы доверяете и на самом деле хотите визуализировать, вы должны прямо сказать, что это безопасно:

```
<p class="{{ class_name|safe }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

Если у вас есть блок, содержащий несколько переменных, которые являются безопасными, вы можете локально отключить автоматическое экранирование:

```
{% autoescape off %}
<p class="{{ class_name }}">{{ paragraph }}</p>
{% endautoescape %}
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px"><script>alert('hello world!');</script></p>
```

Вы также можете пометить строку как безопасную вне шаблона:

```
from django.utils.safestring import mark_safe

context = {
    'class_name': 'large" style="font-size:4000px',
```

```
    'paragraph': mark_safe(
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&amp;quot; style=&amp;quot;font-size: 4000px"><script>alert('hello
world!');</script></p>
```

Некоторые утилиты Django, такие как `format_html` уже возвращают строки, помеченные как безопасные:

```
from django.utils.html import format_html

context = {
    'var': format_html('<b>{}</b> {}', 'hello', '<i>world!</i>'),
}
```

```
<p>{{ var }}</p>
<!-- Will be rendered as -->
<p><b>hello</b> &lt;i&gt;world!&lt;/i&gt;</p>
```

Защита от перегрузки

> Clickjacking - это вредоносная техника обмана веб-пользователя, нажатие на что-то отличное от того, что пользователь видит, на который они нажимают.
> Учить больше

Чтобы включить защиту от `XFrameOptionsMiddleware` , добавьте `XFrameOptionsMiddleware` в свои классы промежуточного программного обеспечения. Это уже должно быть, если вы не удалили его.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
]
```

Это промежуточное программное обеспечение устанавливает заголовок «X-Frame-Options» для всех ваших ответов, если явно не освобождено или уже установлено (не переопределено, если оно уже установлено в ответе). По умолчанию установлено значение «SAMEORIGIN». Чтобы изменить это, используйте настройку `X_FRAME_OPTIONS` :

```
X_FRAME_OPTIONS = 'DENY'
```

Вы можете переопределить поведение по умолчанию для каждого представления.

```
from django.utils.decorators import method_decorator
```

```
from django.views.decorators.clickjacking import (
    xframe_options_exempt, xframe_options_deny, xframe_options_sameorigin,
)

xframe_options_exempt_m = method_decorator(xframe_options_exempt, name='dispatch')

@xframe_options_sameorigin
def my_view(request, *args, **kwargs):
    """Forces 'X-Frame-Options: SAMEORIGIN'."""
    return HttpResponse(...)

@method_decorator(xframe_options_deny, name='dispatch')
class MyView(View):
    """Forces 'X-Frame-Options: DENY'."""

@xframe_options_exempt_m
class MyView(View):
    """Does not set 'X-Frame-Options' header when passing through the
    XFrameOptionsMiddleware.
    """
```

Защита от перекрестных ссылок (CSRF)

> Подделка запросов на межсайтовый запрос, также известный как атака одним нажатием или сеансом верховой езды и сокращенная как CSRF или XSRF, является видом вредоносного использования веб-сайта, на котором неавторизованные команды передаются от пользователя, которому доверяет веб-сайт. Учить больше

Чтобы включить защиту CSRF, добавьте `CsrfViewMiddleware` в классы промежуточного программного обеспечения. Это промежуточное ПО включено по умолчанию.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.csrf.CsrfViewMiddleware',
    ...
]
```

Это промежуточное программное обеспечение будет устанавливать маркер в cookie исходящего ответа. Всякий раз, когда входящий запрос использует небезопасный метод (любой метод, кроме `GET` , `HEAD` , `OPTIONS` и `TRACE`), cookie должен соответствовать `csrfmiddlewaretoken` который отправляется как `csrfmiddlewaretoken` формы `csrfmiddlewaretoken` или как заголовок `X-CsrfToken` . Это гарантирует, что клиент, инициирующий запрос, также является владельцем файла cookie и, как правило, сеансом (аутентифицированным).

Если запрос выполняется через `HTTPS` , включена строгая проверка ссылок. Если заголовок `HTTP_REFERER` не соответствует хосту текущего запроса или хоста в `CSRF_TRUSTED_ORIGINS` (новый в `CSRF_TRUSTED_ORIGINS` 1.9), запрос `CSRF_TRUSTED_ORIGINS` .

Формы, использующие метод `POST` должны включать токен CSRF в шаблоне. `{% csrf_token`

`%}` шаблона `{% csrf_token %}` выведет скрытое поле и будет гарантировать, что cookie будет установлен в ответ:

```
<form method='POST'>
{% csrf_token %}
...
</form>
```

Отдельные представления, которые не уязвимы для атак CSRF, могут быть освобождены с `@csrf_exempt` декоратора `@csrf_exempt` :

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def my_view(request, *args, **kwargs):
    """Allows unsafe methods without CSRF protection"""
    return HttpResponse(...)
```

Хотя это и не рекомендуется, вы можете отключить `CsrfViewMiddleware` если многие ваши взгляды не уязвимы для атак CSRF. В этом случае вы можете использовать декоратор `@csrf_protect` для защиты отдельных видов:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request, *args, **kwargs):
    """This view is protected against CSRF attacks if the middleware is disabled"""
    return HttpResponse(...)
```

Прочитайте Безопасность онлайн: https://riptutorial.com/ru/django/topic/2957/безопасность

глава 16: Виджеты форм

Examples

Простой текстовый входной виджет

Самый простой пример виджета - пользовательский ввод текста. Например, чтобы создать `<input type="tel">` , вы должны подклассифицировать `TextInput` и установить `input_type` в `'tel'` .

```
from django.forms.widgets import TextInput

class PhoneInput(TextInput):
    input_type = 'tel'
```

Композитный виджет

Вы можете создавать виджеты, состоящие из нескольких виджетов с помощью `MultiWidget` .

```
from datetime import date

from django.forms.widgets import MultiWidget, Select
from django.utils.dates import MONTHS

class SelectMonthDateWidget(MultiWidget):
    """This widget allows the user to fill in a month and a year.

    This represents the first day of this month or, if `last_day=True`, the
    last day of this month.
    """

    default_nb_years = 10

    def __init__(self, attrs=None, years=None, months=None, last_day=False):
        self.last_day = last_day

        if not years:
            this_year = date.today().year
            years = range(this_year, this_year + self.default_nb_years)
        if not months:
            months = MONTHS

        # Here we will use two `Select` widgets, one for months and one for years
        widgets = (Select(attrs=attrs, choices=months.items()),
                   Select(attrs=attrs, choices=((y, y) for y in years)))
        super().__init__(widgets, attrs)

    def format_output(self, rendered_widgets):
        """Concatenates rendered sub-widgets as HTML"""
        return (
            '<div class="row">'
            '<div class="col-xs-6">{}</div>'
            '<div class="col-xs-6">{}</div>'
```

```
            '</div>'
        ).format(*rendered_widgets)

    def decompress(self, value):
        """Split the widget value into subwidgets values.
        We expect value to be a valid date formated as `%Y-%m-%d`.
        We extract month and year parts from this string.
        """
        if value:
            value = date(*map(int, value.split('-')))
            return [value.month, value.year]
        return [None, None]

    def value_from_datadict(self, data, files, name):
        """Get the value according to provided `data` (often from `request.POST`)
        and `files` (often from `request.FILES`, not used here)
        `name` is the name of the form field.

        As this is a composite widget, we will grab multiple keys from `data`.
        Namely: `field_name_0` (the month) and `field_name_1` (the year).
        """
        datalist = [
            widget.value_from_datadict(data, files, '{}_{}'.format(name, i))
            for i, widget in enumerate(self.widgets)]
        try:
            # Try to convert it as the first day of a month.
            d = date(day=1, month=int(datelist[0]), year=int(datelist[1]))
            if self.last_day:
                # Transform it to the last day of the month if needed
                if d.month == 12:
                    d = d.replace(day=31)
                else:
                    d = d.replace(month=d.month+1) - timedelta(days=1)
        except (ValueError, TypeError):
            # If we failed to recognize a valid date
            return ''
        else:
            # Convert it back to a string with format `%Y-%m-%d`
            return str(d)
```

Прочитайте Виджеты форм онлайн: https://riptutorial.com/ru/django/topic/1230/виджеты-форм

глава 17: Выражения F ()

Вступление

Выражение F () - это способ Django использовать объект Python для ссылки на значение поля модели или аннотированного столбца в базе данных без необходимости вытягивать значение в память Python. Это позволяет разработчикам избегать определенных условий гонки, а также фильтровать результаты на основе значений полей модели.

Синтаксис

- из django.db.models import F

Examples

Избежать условий гонки

См. Этот вопрос Q & A, если вы не знаете, какие условия гонки.

Следующий код может подпадать под условия гонки:

```
article = Article.objects.get(pk=69)
article.views_count += 1
article.save()
```

Если `views_count` равен `1337` , это приведет к такому запросу:

```
UPDATE app_article SET views_count = 1338 WHERE id=69
```

Если два клиента одновременно обращаются к этой статье, *может* произойти, что второй HTTP-запрос выполняет `Article.objects.get(pk=69)` прежде чем первый выполнит `article.save()` . Таким образом, оба запроса будут иметь `views_count = 1337` , увеличивают его и сохраняют `views_count = 1338` в базе данных, тогда как на самом деле это должно быть `1339` .

Чтобы исправить это, используйте выражение `F()` :

```
article = Article.objects.get(pk=69)
article.views_count = F('views_count') + 1
article.save()
```

Это, с другой стороны, приведет к такому запросу:

```
UPDATE app_article SET views_count = views_count + 1 WHERE id=69
```

Обновление набора запросов навалом

Предположим, что мы хотим удалить 2 upvotes из всех статей автора с идентификатором `51` .
Выполнение этого только с Python будет выполнять `N` запросов (`N` - количество статей в наборе запросов):

```
for article in Article.objects.filter(author_id=51):
    article.upvotes -= 2
    article.save()
    # Note that there is a race condition here but this is not the focus
    # of this example.
```

Что делать, если вместо того, чтобы вытаскивать все статьи в Python, перебирать их, уменьшать upvotes и сохранять каждый обновленный обратно в базу данных, был другой способ?
Используя выражение `F()` , можно сделать это в одном запросе:

```
Article.objects.filter(author_id=51).update(upvotes=F('upvotes') - 2)
```

Что можно перевести в следующем SQL-запросе:

```
UPDATE app_article SET upvotes = upvotes - 2 WHERE author_id = 51
```

Почему это лучше?

- Вместо того, чтобы Python выполнял работу, мы передаем нагрузку в базу данных, которая настроена так, чтобы делать такие запросы.
- Эффективно сокращает количество запросов к базе данных, необходимых для достижения желаемого результата.

Выполнение арифметических операций между полями

Выражения `F()` могут использоваться для выполнения арифметических операций (`+` , `-` , `*` и т. Д.) Среди полей модели, чтобы определить алгебраический поиск / связь между ними.

- Пусть модель будет:

  ```
  class MyModel(models.Model):
      int_1 = models.IntegerField()
      int_2 = models.IntegerField()
  ```

- Теперь давайте предположим , что мы хотим получить все объекты `MyModel` таблицы Кто на `int_1` и `int_2` поля удовлетворяют этому уравнению: `int_1 + int_2 >= 5` .

Используя `annotate()` и `filter()` получаем:

```
result = MyModel.objects.annotate(
             diff=F(int_1) + F(int_2)
         ).filter(diff__gte=5)
```

`result` теперь содержит все вышеупомянутые объекты.

Хотя в этом примере используются поля `Integer` , этот метод будет работать во всех областях, в которых может быть применена арифметическая операция.

Прочитайте Выражения F () онлайн: https://riptutorial.com/ru/django/topic/2765/выражения-f---

глава 18: Джанго-фильтр

Examples

Использование django-фильтра с CBV

`django-filter` - это общая система фильтрации Django QuerySets на основе пользовательских настроек. Документация использует его в функциональном представлении как модель продукта:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

Фильтр будет выглядеть следующим образом:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')

    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

Чтобы использовать это в CBV, переопределите `get_queryset()` ListView, а затем верните фильтрованный `querset` :

```
from django.views.generic import ListView
from .filters import ProductFilter

class ArticleListView(ListView):
    model = Product

    def get_queryset(self):
        qs = self.model.objects.all()
        product_filtered_list = ProductFilter(self.request.GET, queryset=qs)
        return product_filtered_list.qs
```

Можно получить доступ к фильтрованным объектам в ваших представлениях, например, с `f.qs` по `f.qs` , в `f.qs` Это приведет к разбивке списка фильтрованных объектов.

Прочитайте Джанго-фильтр онлайн: https://riptutorial.com/ru/django/topic/6101/джанго-фильтр

глава 19: Запуск сельдерея с супервизором

Examples

Конфигурация сельдерея

СЕЛЬДЕРЕЙ

1. Установка - `pip install django-celery`

2. добавлять

3. Основная структура проекта.

```
- src/
  - bin/celery_worker_start # will be explained later on
  - logs/celery_worker.log
  - stack/__init __.py
  - stack/celery.py
  - stack/settings.py
  - stack/urls.py
  - manage.py
```

4. Добавьте файл `celery.py` в `stack/stack/` папку `stack/stack/` папку.

```
from __future__ import absolute_import
import os
from celery import Celery
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'stack.settings')
from django.conf import settings  # noqa
app = Celery('stack')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

5. в ваш `stack/stack/__init__.py` добавьте следующий код:

```
from __future__ import absolute_import
from .celery import app as celery_app  # noqa
```

6. Создайте задачу и отметьте ее, например, как `@shared_task()`

```
@shared_task()
def add(x, y):
    print("x*y={}".format(x*y))
```

7. Запуск работника сельдерея «вручную»:

`celery -A stack worker -l info` если вы также хотите добавить

Управляющий супервайзер

1. Создайте скрипт, чтобы начать работу с сельдереем. Вставьте скрипт в свое приложение. Например: `stack/bin/celery_worker_start`

```
#!/bin/bash

NAME="StackOverflow Project - celery_worker_start"

PROJECT_DIR=/home/stackoverflow/apps/proj/proj/
ENV_DIR=/home/stackoverflow/apps/proj/env/

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd "${PROJECT_DIR}"

if [ -d "${ENV_DIR}" ]
then
    . "${ENV_DIR}bin/activate"
fi

celery -A stack --loglevel='INFO'
```

2. Добавить права выполнения для вновь созданного скрипта:

```
chmod u+x bin/celery_worker_start
```

3. Установите диспетчер (пропустите этот тест, если супервизор уже установлен)

```
apt-get install supervisor
```

4. Добавьте файл конфигурации для своего супервизора, чтобы начать сельдерей. Поместите его в `/etc/supervisor/conf.d/stack_supervisor.conf`

```
[program:stack-celery-worker]
command = /home/stackoverflow/apps/stack/src/bin/celery_worker_start
user = polsha
stdout_logfile = /home/stackoverflow/apps/stack/src/logs/celery_worker.log
redirect_stderr = true
environment = LANG = en_US.UTF-8,LC_ALL = en_US.UTF-8
numprocs = 1
autostart = true
autorestart = true
startsecs = 10
stopwaitsecs = 600
priority = 998
```

5. Перечитать и обновить супервизор

```
sudo supervisorctl reread
    stack-celery-worker: available
```

```
sudo supervisorctl update
    stack-celery-worker: added process group
```

6. Основные команды

```
sudo supervisorctl status stack-celery-worker
    stack-celery-worker      RUNNING    pid 18020, uptime 0:00:50
sudo supervisorctl stop stack-celery-worker
    stack-celery-worker: stopped
sudo supervisorctl start stack-celery-worker
    stack-celery-worker: started
sudo supervisorctl restart stack-celery-worker
    stack-celery-worker: stopped
    stack-celery-worker: started
```

Celery + RabbitMQ с супервизором

Сельдерей требует брокера для обработки сообщений. Мы используем RabbitMQ, потому что его легко настроить, и он хорошо поддерживается.

Установите rabbitmq, используя следующую команду:

```
sudo apt-get install rabbitmq-server
```

По завершении установки создайте пользователя, добавьте виртуальный хост и установите разрешения.

```
sudo rabbitmqctl add_user myuser mypassword
sudo rabbitmqctl add_vhost myvhost
sudo rabbitmqctl set_user_tags myuser mytag
sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

Чтобы запустить сервер:

```
sudo rabbitmq-server
```

Мы можем установить сельдерей с пипеткой:

```
pip install celery
```

В файле настроек Django settings.py ваш URL-адрес брокера будет выглядеть примерно так:

```
BROKER_URL = 'amqp://myuser:mypassword@localhost:5672/myvhost'
```

Теперь начните работу сельдерея

```
celery -A your_app worker -l info
```

Эта команда запускает работника Celery для выполнения любых задач, определенных в вашем приложении django.

Supervisor - это программа Python, которая позволяет вам контролировать и поддерживать любые процессы unix. Он также может перезапускать разбитые процессы. Мы используем его, чтобы убедиться, что работники сельдерея всегда работают.

Во-первых, Install supervisor

```
sudo apt-get install supervisor
```

Создайте файл your_proj.conf в вашем супервизоре conf.d (/etc/supervisor/conf.d/your_proj.conf):

```
[program:your_proj_celery]
command=/home/your_user/your_proj/.venv/bin/celery --app=your_proj.celery:app worker -l info
directory=/home/your_user/your_proj
numprocs=1
stdout_logfile=/home/your_user/your_proj/logs/celery-worker.log
stderr_logfile=/home/your_user/your_proj/logs/low-worker.log
autostart=true
autorestart=true
startsecs=10
```

После создания и сохранения нашего конфигурационного файла мы можем сообщить Супервизору нашей новой программы с помощью команды supervisorctl. Сначала мы советуем Supervisor искать любые новые или измененные конфигурации программ в каталоге /etc/supervisor/conf.d с:

```
sudo supervisorctl reread
```

Затем следуйте указаниям, чтобы внести изменения в:

```
sudo supervisorctl update
```

Когда наши программы будут запущены, несомненно, будет время, когда мы хотим остановить, перезапустить или увидеть их статус.

```
sudo supervisorctl status
```

Для перезапуска вашего экземпляра сельдерея:

```
sudo supervisorctl restart your_proj_celery
```

Прочитайте Запуск сельдерея с супервизором онлайн:
https://riptutorial.com/ru/django/topic/7091/запуск-сельдерея-с-супервизором

глава 20: интернационализация

Синтаксис

- Gettext (сообщение)
- ngettext (единственное число, число, число)
- ugettext (сообщение)
- ungettext (единственное число, множественное число, число)
- pgettext (контекст, сообщение)
- npgettext (контекст, единственное число, множественное число, число)
- gettext_lazy (сообщение)
- ngettext_lazy (единственное число, множественное число, число = нет)
- ugettext_lazy (сообщение)
- ungettext_lazy (единственное число, множественное число, число = нет)
- pgettext_lazy (контекст, сообщение)
- npgettext_lazy (контекст, единственное число, множественное число, число = нет)
- gettext_noop (сообщение)
- ugettext_noop (сообщение)

Examples

Введение в интернационализацию

Настройка

settings.py

```
from django.utils.translation import ugettext_lazy as _

USE_I18N = True  # Enable Internationalization
LANGUAGE_CODE = 'en'  # Language in which original texts are written
LANGUAGES = [  # Available languages
    ('en', _("English")),
    ('de', _("German")),
    ('fr', _("French")),
]

# Make sure the LocaleMiddleware is included, AFTER SessionMiddleware
# and BEFORE middlewares using internationalization (such as CommonMiddleware)
MIDDLEWARE_CLASSES = [
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
]
```

Маркировка строк как переводимых

Первым шагом в переводе является *маркировка строк как переводимых* . Это передается через одну из функций `gettext` (см. Раздел « Синтаксис»). Например, вот пример модели:

```
from django.utils.translation import ugettext_lazy as _
# It is common to import gettext as the shortcut `_` as it is often used
# several times in the same file.

class Child(models.Model):

    class Meta:
        verbose_name = _("child")
        verbose_name_plural = _("children")

    first_name = models.CharField(max_length=30, verbose_name=_("first name"))
    last_name = models.CharField(max_length=30, verbose_name=_("last name"))
    age = models.PositiveSmallIntegerField(verbose_name=_("age"))
```

Все строки, заключенные в `_()` , теперь помечены как переводимые. При печати они всегда будут отображаться в виде инкапсулированной строки независимо от выбранного языка (так как перевод еще не доступен).

Перевод строк

Этот пример достаточно для начала перевода. В большинстве случаев вы хотите только отметить строки как переведенные, чтобы **предвидеть предполагаемую интернационализацию** вашего проекта. Таким образом, это рассматривается в другом примере .

Перевод с ленивого на нежирный

При использовании нелазного перевода строки сразу переводятся.

```
>>> from django.utils.translation import activate, ugettext as _
>>> month = _("June")
>>> month
'June'
>>> activate('fr')
>>> _("June")
'juin'
>>> activate('de')
>>> _("June")
'Juni'
>>> month
'June'
```

При использовании лень перевод происходит только при фактическом использовании.

```
>>> from django.utils.translation import activate, ugettext_lazy as _
>>> month = _("June")
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> str(month)
'June'
>>> activate('fr')
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> "month: {}".format(month)
'month: juin'
>>> "month: %s" % month
'month: Juni'
```

Вы должны использовать ленивый перевод в тех случаях, когда:

- Перевод не может быть активирован (язык не выбран), когда `_("some string")` оценивается
- Некоторые строки могут оцениваться только при запуске (например, в атрибутах класса, таких как определения полей модели и формы)

Перевод в шаблоны

Чтобы включить перевод в шаблонах, вы должны загрузить библиотеку `i18n` .

```
{% load i18n %}
```

Основной перевод выполняется с тегом `trans` template.

```
{% trans "Some translatable text" %}
{# equivalent to python `ugettext("Some translatable text")` #}
```

Тег `trans` template поддерживает контекст:

```
{% trans "May" context "month" %}
{# equivalent to python `pgettext("May", "month")` #}
```

Чтобы включить заполнители в строку перевода, как в:

```
_("My name is {first_name} {last_name}").format(first_name="John", last_name="Doe")
```

Вам нужно будет использовать `blocktrans` шаблона `blocktrans` :

```
{% blocktrans with first_name="John" last_name="Doe" %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Конечно, вместо `"John"` и `"Doe"` вас могут быть переменные и фильтры:

```
{% blocktrans with first_name=user.first_name last_name=user.last_name|title %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Если `first_name` и `last_name` уже находятся в вашем контексте, вы можете даже опустить предложение `with` :

```
{% blocktrans %}My name is {{ first_name }} {{ last_name }}{% endblocktrans %}
```

Однако могут использоваться только контекстные переменные верхнего уровня. **Это не будет работать:**

```
{% blocktrans %}
    My name is {{ user.first_name }} {{ user.last_name }}
{% endblocktrans %}
```

Это происходит главным образом потому, что имя переменной используется в качестве заполнителя в файлах перевода.

`blocktrans` шаблона `blocktrans` также `blocktrans` плюрализацию.

```
{% blocktrans count nb=users|length }}
    There is {{ nb }} user.
{% plural %}
    There are {{ nb }} users.
{% endblocktrans %}
```

Наконец, независимо от библиотеки `i18n` , вы можете передавать переводимые строки в теги шаблонов с помощью синтаксиса `_("")` .

```
{{ site_name|default:_("It works!") }}
{% firstof var1 var2 _("translatable fallback") %}
```

Это некоторая волшебная встроенная система шаблонов django для имитации синтаксиса вызова функции, но это не вызов функции. `_("It works!")` Передается тегу шаблона по `default` в виде строки `'_("It works!")'` Которая затем анализируется переводимой строкой, так же, как `name` будет анализироваться как переменная, а `"name"` будет анализируется как строка.

Перевод строк

Чтобы перевести строки, вам придется создавать файлы переводов. Для этого django отправляется с командой управления `makemessages` .

```
$ django-admin makemessages -l fr
processing locale fr
```

Вышеупомянутая команда обнаружит все строки, помеченные как переводимые в ваши установленные приложения, и создаст один языковой файл для каждого приложения для перевода на французский язык. Например, если у вас есть только одно приложение `myapp` содержащее переводимые строки, это создаст файл `myapp/locale/fr/LC_MESSAGES/django.po` . Этот файл может выглядеть так:

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-24 14:01+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: myapp/models.py:22
msgid "user"
msgstr ""

#: myapp/models.py:39
msgid "A user already exists with this email address."
msgstr ""

#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
```

Сначала вы должны заполнить заполнители (подчеркнуты с помощью верхних частей). Затем переведите строки. `msgid` "строка, помеченная как переводимая в вашем коде. `msgstr` , где вы должны написать перевод строки прямо выше.

Когда строка содержит заполнители, вам также придется включить их в свой перевод. Например, вы переведете последнее сообщение следующим образом:

```
#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
"En vous inscrivant, vous acceptez nos <a href=\"%(terms_url)s\" "
"target=_blank>Conditions d'utilisation</a>"
```

Как только ваш файл перевода будет завершен, вам придется скомпилировать файлы `.po` файлы `.mo` . Это делается путем вызова `compilemessages` управления `compilemessages` :

```
$ django-admin compilemessages
```

Вот и все, теперь переводы доступны.

Чтобы обновить ваши файлы переводов при внесении изменений в свой код, вы можете повторно запустить `django-admin makemessages -l fr` . Это будет обновлять файлы `.po` , сохраняя ваши существующие переводы и добавляя новые. Удаленные строки по-прежнему будут доступны в комментариях. Чтобы обновить файлы `.po` для всех языков, запустите `django-admin makemessages -a` . После того, как ваши файлы `.po` будут обновлены, не забудьте снова запустить `django-admin compilemessages` для создания файлов `.mo` .

Случай использования Noop

`(u)gettext_noop` позволяет пометить строку как переводимую, не переведя ее.

Типичным примером использования является то, когда вы хотите зарегистрировать сообщение для разработчиков (на английском языке), но также хотите отобразить его клиенту (на запрошенном языке). **Вы можете передать переменную `gettext` , но ее содержимое не будет обнаружено как переводимая строка, потому что это переменная per definition.** ,

```
# THIS WILL NOT WORK AS EXPECTED
import logging
from django.contrib import messages

logger = logging.getLogger(__name__)

error_message = "Oops, something went wrong!"
logger.error(error_message)
messages.error(request, _(error_message))
```

Сообщение об ошибке не появится в файле `.po` и вам нужно будет запомнить его, чтобы добавить его вручную. Чтобы исправить это, вы можете использовать `gettext_noop` .

```
error_message = ugettext_noop("Oops, something went wrong!")
logger.error(error_message)
messages.error(request, _(error_message))
```

Теперь строка `"Oops, something went wrong!"` будет обнаружен и доступен в файле `.po` при его создании. И ошибка будет по-прежнему регистрироваться на английском языке для разработчиков.

Обычные подводные камни

нечеткие переводы

Иногда `makemessages` может думать, что строка, найденная для перевода, несколько похожа на уже существующий перевод. Он будет отмечать его в файле `.po` специальным `fuzzy` комментарием следующим образом:

```
#: templates/randa/map.html:91
#, fuzzy
msgid "Country"
msgstr "Länderinfo"
```

Даже если перевод верен или вы обновили его, чтобы исправить его, он не будет использоваться для перевода вашего проекта, если вы не удалите строку с `fuzzy` комментариями.

Многострочные строки

`makemessages` анализирует файлы в различных форматах, от простого текста до кода python и не предназначен для следования всем возможным правилам для наличия многострочных строк в этих форматах. Большую часть времени он будет отлично работать с однострочными строками, но если у вас есть такая конструкция:

```
translation = _("firstline"
"secondline"
"thirdline")
```

Это займет только `firstline` для перевода. Решение этого - избегать использования многострочных строк, когда это возможно.

Прочитайте интернационализация онлайн: https://riptutorial.com/ru/django/topic/2579/интернационализация

глава 21: Использование Redis с Django - Caching Backend

замечания

Использование django-redis-cache или django-redis являются эффективными решениями для хранения всех кэшированных элементов. Хотя, конечно, Redis может быть настроен непосредственно как `SESSION_ENGINE` , одной из эффективных стратегий является настройка кэширования (как указано выше) и объявление кеша по умолчанию как `SESSION_ENGINE` . Хотя это действительно тема для другой документальной статьи, ее релевантность приводит к включению.

Просто добавьте следующее в `settings.py` :

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Examples

Использование django-redis-cache

Одной из возможных реализаций Redis в качестве базовой утилиты кэширования является пакет django-redis-cache .

В этом примере предполагается, что у вас уже есть сервер Redis .

```
$ pip install django-redis-cache
```

Отредактируйте файл `settings.py` чтобы включить объект `CACHES` (см. Документацию по кэшированию в Django).

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': 'localhost:6379',
        'OPTIONS': {
            'DB': 0,
        }
    }
}
```

Использование django-redis

Одной из возможных реализаций Redis в качестве базовой утилиты кэширования является

пакет django-redis .

В этом примере предполагается, что у вас уже есть сервер Redis .

```
$ pip install django-redis
```

Отредактируйте файл `settings.py` чтобы включить объект `CACHES` (см. Документацию по кэшированию в Django).

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Прочитайте Использование Redis с Django - Caching Backend онлайн:
https://riptutorial.com/ru/django/topic/4085/использование-redis-с-django---caching-backend

глава 22: Как восстановить миграцию django

Вступление

Когда вы разрабатываете приложение Django, могут возникнуть ситуации, когда вы можете сэкономить много времени, просто очистив и перезагрузив свои миграции.

Examples

Сброс миграции Django: удаление существующей базы данных и миграция в виде свежих

Drop / Delete your database Если вы используете SQLite для своей базы данных, просто удалите этот файл. Если вы используете MySQL / Postgres или любую другую систему баз данных, вам придется отказаться от базы данных, а затем воссоздать новую базу данных.

Теперь вам нужно удалить весь файл миграции, кроме файла «init.py», находящегося внутри папки миграции в папке вашего приложения.

Обычно папка миграции находится по адресу

```
/your_django_project/your_app/migrations
```

Теперь, когда вы удалили базу данных и файл миграции, просто запустите следующие команды, так как вы перенесли первый раз, когда вы создадите проект django.

```
python manage.py makemigrations
python manage.py migrate
```

Прочитайте Как восстановить миграцию django онлайн:
https://riptutorial.com/ru/django/topic/9513/как-восстановить-миграцию-django

глава 23: Как использовать Django с Cookiecutter?

Examples

Установка и настройка проекта django с использованием Cookiecutter

Ниже приведены предварительные условия для установки Cookiecutter:

- зернышко
- virtualenv
- PostgreSQL

Создайте virtualenv для своего проекта и активируйте его:

```
$ mkvirtualenv <virtualenv name>
$ workon <virtualenv name>
```

Теперь установите Cookiecutter, используя:

```
$ pip install cookiecutter
```

Измените каталоги в папку, в которой вы хотите, чтобы ваш проект жил. Теперь выполните следующую команду для создания проекта django:

```
$ cookiecutter https://github.com/pydanny/cookiecutter-django.git
```

Эта команда запускает cookiecutter с репозиторией cookiecutter-django, предлагая нам ввести подробные сведения о конкретном проекте. Нажмите «enter», не набрав ничего, чтобы использовать значения по умолчанию, которые показаны в [скобках] после вопроса.

```
project_name [project_name]: example_project
repo_name [example_project]:
author_name [Your Name]: Atul Mishra
email [Your email]: abc@gmail.com
description [A short description of the project.]: Demo Project
domain_name [example.com]: example.com
version [0.1.0]: 0.1.0
timezone [UTC]: UTC
now [2016/03/08]: 2016/03/08
year [2016]: 2016
use_whitenoise [y]: y
use_celery [n]: n
use_mailhog [n]: n
use_sentry [n]: n
use_newrelic [n]: n
use_opbeat [n]: n
```

```
windows [n]: n
use_python2 [n]: n
```

Более подробную информацию о вариантах создания проекта можно найти в официальной документации. Проект теперь настроен.

Прочитайте Как использовать Django с Cookiecutter? онлайн: https://riptutorial.com/ru/django/topic/5385/как-использовать-django-с-cookiecutter-

глава 24: Команды управления

Вступление

Команды управления - это мощные и гибкие скрипты, которые могут выполнять действия в вашем проекте Django или в базе данных. В дополнение к различным командам по умолчанию можно писать самостоятельно!

По сравнению с обычными сценариями Python, использование фреймворка командной консоли означает, что некоторые утомительные работы по настройке автоматически выполняются для вас за кулисами.

замечания

Команды управления могут быть вызваны либо из:

- `django-admin <command> [options]`
- `python -m django <command> [options]`
- `python manage.py <command> [options]`
- `./manage.py <command> [options]` если у manage.py есть разрешения на выполнение (`chmod +x manage.py`)

Чтобы использовать команды управления с помощью Cron:

```
*/10 * * * * pythonuser /var/www/dev/env/bin/python /var/www/dev/manage.py <command> [options]
> /dev/null
```

Examples

Создание и запуск команды управления

Для выполнения действий в Django с использованием командной строки или других служб (где пользователь / запрос не используется), вы можете использовать `management commands` .

Модули Django могут быть импортированы по мере необходимости.

Для каждой команды необходимо создать отдельный файл:
`myapp/management/commands/my_command.py`
(Каталоги `management` и `commands` должны иметь пустой файл __init__.py)

```
from django.core.management.base import BaseCommand, CommandError

# import additional classes/modules as needed
# from myapp.models import Book
```

```
class Command(BaseCommand):
    help = 'My custom django management command'

    def add_arguments(self, parser):
       parser.add_argument('book_id', nargs='+', type=int)
       parser.add_argument('author' , nargs='+', type=str)

    def handle(self, *args, **options):
       bookid = options['book_id']
       author = options['author']
       # Your code goes here

        # For example:
        # books = Book.objects.filter(author="bob")
        # for book in books:
        #    book.name = "Bob"
        #    book.save()
```

Здесь обязателен класс name **Command,** который расширяет **BaseCommand** или один из его подклассов.

Имя команды управления - это имя файла, содержащего его. Чтобы запустить команду в приведенном выше примере, используйте следующую команду в каталоге проекта:

```
python manage.py my_command
```

> Обратите внимание, что запуск команды может занять несколько секунд (из-за импорта модулей). Поэтому в некоторых случаях рекомендуется создавать процессы `daemon` вместо `management commands` .

Подробнее о командах управления

Получить список существующих команд

Вы можете получить список доступных команд следующим образом:

```
>>> python manage.py help
```

Если вы не понимаете какую-либо команду или ищете дополнительные аргументы, вы можете использовать аргумент **-h,** как это

```
>>> python manage.py command_name -h
```

Здесь command_name будет вашим именем команды желания, это покажет вам текст помощи из команды.

```
>>> python manage.py runserver -h
>>> usage: manage.py runserver [-h] [--version] [-v {0,1,2,3}]
                              [--settings SETTINGS] [--pythonpath PYTHONPATH]
                              [--traceback] [--no-color] [--ipv6] [--nothreading]
```

```
                            [--noreload] [--nostatic] [--insecure]
                            [addrport]

Starts a lightweight Web server for development and also serves static files.

positional arguments:
  addrport              Optional port number, or ipaddr:port

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v {0,1,2,3}, --verbosity {0,1,2,3}
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on CommandError exceptions
  --no-color            Don't colorize the command output.
  --ipv6, -6            Tells Django to use an IPv6 address.
  --nothreading         Tells Django to NOT use threading.
  --noreload            Tells Django to NOT use the auto-reloader.
  --nostatic            Tells Django to NOT automatically serve static files
                        at STATIC_URL.
  --insecure            Allows serving static files even if DEBUG is False.
```

Список доступных команд

Использование django-admin вместо manage.py

Вы можете избавиться от `manage.py` и вместо этого использовать команду `django-admin` . Для этого вам нужно вручную сделать то, что `manage.py` делает:

- Добавьте свой путь к вашему PYTHONPATH
- Установите DJANGO_SETTINGS_MODULE

```
export PYTHONPATH="/home/me/path/to/your_project"
export DJANGO_SETTINGS_MODULE="your_project.settings"
```

Это особенно полезно в virtualenv, где вы можете установить эти переменные среды в сценарии `postactivate` .

Преимущество команды `django-admin` в том, что вы находитесь там, где находитесь в своей файловой системе.

Встроенные команды управления

Django поставляется с несколькими встроенными командами управления, используя `python manage.py [command]` или, когда у manage.py есть + x (исполняемые) права просто `./manage.py`

`[command]` . Ниже приведены некоторые из наиболее часто используемых:

Получить список всех доступных команд

```
./manage.py help
```

Запустите сервер Django на localhost: 8000; необходимо для локального тестирования

```
./manage.py runserver
```

Запустите консоль python (или ipython if installed) с предустановленными настройками Django вашего проекта (попытка получить доступ к частям вашего проекта на терминале python без этого не удастся).

```
./manage.py shell
```

Создайте новый файл миграции базы данных на основе изменений, внесенных вами в ваши модели. См. Миграции

```
./manage.py makemigrations
```

Примените любые непримененные миграции к текущей базе данных.

```
./manage.py migrate
```

Запустите тестовый пакет вашего проекта. См. Unit Testing

```
./manage.py test
```

Возьмите все статические файлы вашего проекта и `STATIC_ROOT` их в папку, указанную в `STATIC_ROOT` чтобы их можно было подавать на производство.

```
./manage.py collectstatic
```

Разрешить создание суперпользователя.

```
./manage.py createsuperuser
```

Измените пароль указанного пользователя.

```
./manage.py changepassword username
```

Полный список доступных команд

Прочитайте Команды управления онлайн: https://riptutorial.com/ru/django/topic/1661/команды-

управления

глава 25: Контекстные процессоры

замечания

Используйте контекстные процессоры для добавления переменных, доступных в любом месте шаблонов.

Укажите функцию или функции, которые возвращают `dict` s переменных, которые вы хотите, затем добавьте эти функции в `TEMPLATE_CONTEXT_PROCESSORS` .

Examples

Используйте контекстный процессор для доступа к параметрам. DEBUG в шаблонах

В `myapp/context_processors.py` :

```
from django.conf import settings

def debug(request):
  return {'DEBUG': settings.DEBUG}
```

В `settings.py` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.debug',
            ],
        },
    },
]
```

или, для версий <1.9:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'myapp.context_processors.debug',
)
```

Тогда в моих шаблонах просто:

```
 {% if DEBUG %} .header { background:#f00; } {% endif %}
 {{ DEBUG }}
```

Использование контекстного процессора для доступа к вашим последним блогам во всех шаблонах

Предполагая, что у вас есть модель под названием `Post` определенная в файле `models.py` который содержит сообщения в блоге, и имеет поле `date_published` .

Шаг 1: Запишите обработчик контекста

Создайте (или добавьте) файл в каталог приложения, называемый `context_processors.py` :

```
from myapp.models import Post

def recent_blog_posts(request):
    return {'recent_posts':Post.objects.order_by('-date_published')[0:3],}  # Can change
numbers for more/fewer posts
```

Шаг 2. Добавьте обработчик контекста в ваш файл настроек.

Убедитесь, что вы добавили новый процессор контекста в файл `settings.py` в переменной `TEMPLATES` :

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.recent_blog_posts',
            ],
        },
    },
]
```

(В версиях Django до 1.9 это было задано непосредственно в `settings.py` с помощью переменной `TEMPLATE_CONTEXT_PROCESSORS` .)

Шаг 3. Использование контекстного процессора в ваших шаблонах

Не нужно передавать последние записи в блоге через отдельные просмотры больше! Просто используйте `recent_blog_posts` в любом шаблоне.

Например, в `home.html` вы можете создать боковую панель со ссылками на последние сообщения:

```
<div class="blog_post_sidebar">
    {% for post in recent_blog_posts %}
        <div class="post">
            <a href="{{post.get_absolute_url}}">{{post.title}}</a>
```

```
        </div>
    {% endfor %}
</div>
```

Или в `blog.html` вы можете создать более подробное отображение каждого сообщения:

```
<div class="content">
    {% for post in recent_blog_posts %}
        <div class="post_detail">
            <h2>{{post.title}}</h2>
            <p>Published on {{post.date_published}}</p>
            <p class="author">Written by: {{post.author}}</p>
            <p><a href="{{post.get_absolute_url}}">Permalink</a></p>
            <p class="post_body">{{post.body}}</p>
        </div>
    {% endfor %}
</div>
```

Расширение шаблонов

Контекстный процессор для определения шаблона на основе членства в группе (или любого запроса / логики). Это позволяет нашим публичным / обычным пользователям получать один шаблон и нашу специальную группу, чтобы получить другую.

MyApp / context_processors.py

```
def template_selection(request):
    site_template = 'template_public.html'
    if request.user.is_authenticated():
        if request.user.groups.filter(name="some_group_name").exists():
            site_template = 'template_new.html'

    return {
        'site_template': site_template,
    }
```

Добавьте контекстный процессор в свои настройки.

В ваших шаблонах используйте переменную, определенную в обработчике контекста.

```
{% extends site_template %}
```

Прочитайте Контекстные процессоры онлайн: https://riptutorial.com/ru/django/topic/491/контекстные-процессоры

глава 26: логирование

Examples

Вход в систему Syslog

Можно настроить Django для вывода журнала на локальную или удаленную службу syslog. Эта конфигурация использует встроенный python SysLogHandler .

```
from logging.handlers import SysLogHandler
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format' : "[YOUR PROJECT NAME] [%(asctime)s] %(levelname)s [%(name)s:%(lineno)s]
%(message)s",
            'datefmt' : "%d/%b/%Y %H:%M:%S"
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
        'syslog': {
            'class': 'logging.handlers.SysLogHandler',
            'formatter': 'standard',
            'facility': 'user',
            # uncomment next line if rsyslog works with unix socket only (UDP reception
disabled)
            #'address': '/dev/log'
        }
    },
    'loggers': {
        'django':{
            'handlers': ['syslog'],
            'level': 'INFO',
            'disabled': False,
            'propagate': True
        }
    }
}

# loggers for my apps, uses INSTALLED_APPS in settings
# each app must have a configured logger
# level can be changed as desired: DEBUG, INFO, WARNING...
MY_LOGGERS = {}
for app in INSTALLED_APPS:
    MY_LOGGERS[app] = {
        'handlers': ['syslog'],
        'level': 'DEBUG',
        'propagate': True,
    }
LOGGING['loggers'].update(MY_LOGGERS)
```

Конфигурация основного журнала Django

Внутри Django использует систему регистрации Python. Существует много способов настроить ведение журнала проекта. Вот база:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': "[%(asctime)s] %(levelname)s [%(name)s:%(lineno)s] %(message)s",
            'datefmt': "%Y-%m-%d %H:%M:%S"
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'default'
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'INFO',
        },
    }
}
```

Форматтеры

Он может использоваться для настройки появления журналов, когда они печатаются для вывода. Вы можете определить множество форматировщиков, установив строку ключа для каждого другого форматирования. При объявлении обработчика используется форматер.

Обработчики

Может использоваться для настройки, где будут печататься журналы. В приведенном выше примере они отправляются на stdout и stderr. Существуют различные классы обработчиков:

```
'rotated_logs': {
    'class': 'logging.handlers.RotatingFileHandler',
    'filename': '/var/log/my_project.log',
    'maxBytes': 1024 * 1024 * 5,  # 5 MB
    'backupCount': 5,
    'formatter': 'default'
    'level': 'DEBUG',
},
```

Это приведет к созданию журналов в файле, завершенном по `filename` . В этом примере будет создан новый файл журнала, когда текущий размер достигнет 5 МБ (старый

переименован в my_project.log.1), и последние 5 файлов будут сохранены для архива.

```
'mail_admins': {
    'level': 'ERROR',
    'class': 'django.utils.log.AdminEmailHandler'
},
```

Это отправит каждый журнал `ADMINS` пользователям, указанным в переменной настройки `ADMINS` . Уровень установлен на `ERROR` , поэтому только журналы с уровнем `ERROR` будут отправлены по электронной почте. Это чрезвычайно полезно, чтобы быть в курсе потенциальных ошибок 50x на сервере производства.

Другие обработчики могут использоваться с Django. Для получения полного списка, пожалуйста, ознакомьтесь с соответствующей документацией . Подобно formatters, вы можете определить много обработчиков в одном проекте, установив для каждой другой ключевой строки. Каждый обработчик может использоваться в конкретном регистраторе.

Лесорубы

В `LOGGING` последняя часть настраивает для каждого модуля минимальный уровень ведения журнала, используемые обработчики (ы) и т. Д.

Прочитайте логирование онлайн: https://riptutorial.com/ru/django/topic/1231/логирование

глава 27: Маршрутизаторы баз данных

Examples

Добавление файла маршрутизации базы данных

Чтобы использовать несколько баз данных в Django, просто укажите их в `settings.py` :

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'django_db_user',
        'PASSWORD': os.environ['LOCAL_DB_PASSWORD']
    },
    'users': {
        'NAME': 'remote_data',
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'remote.host.db',
        'USER': 'remote_user',
        'PASSWORD': os.environ['REMOTE_DB_PASSWORD']
    }
}
```

Используйте файл `dbrouters.py` чтобы указать, какие модели должны работать с базами данных для каждого класса операций с базой данных, например, для удаленных данных, хранящихся в `remote_data` , может `remote_data` следующее:

```
class DbRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read remote models go to remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def db_for_write(self, model, **hints):
        """
        Attempts to write remote models go to the remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Do not allow relations involving the remote database
        """
```

```
        if obj1._meta.app_label == 'remote' or \
           obj2._meta.app_label == 'remote':
           return False
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Do not allow migrations on the remote database
        """
        if model._meta.app_label == 'remote':
            return False
        return True
```

Наконец, добавьте `dbrouter.py` в `settings.py` :

```
DATABASE_ROUTERS = ['path.to.DbRouter', ]
```

Указание различных баз данных в коде

`obj.save()` метод `obj.save()` будет использовать базу данных по умолчанию, или если используется маршрутизатор базы данных, он будет использовать базу данных, как указано в `db_for_write` . Вы можете переопределить его, используя:

```
obj.save(using='other_db')
obj.delete(using='other_db')
```

Аналогично, для чтения:

```
MyModel.objects.using('other_db').all()
```

Прочитайте Маршрутизаторы баз данных онлайн: https://riptutorial.com/ru/django/topic/3395/маршрутизаторы-баз-данных

глава 28: Маршрутизация URL

Examples

Как Django обрабатывает запрос

Django обрабатывает запрос путем маршрутизации входящего URL-пути к функции просмотра. Функция просмотра отвечает за возврат ответа обратно клиенту, выполняющему запрос. Различные URL-адреса обычно обрабатываются различными функциями просмотра. Чтобы перенаправить запрос на определенную функцию просмотра, Django просматривает вашу конфигурацию URL (или URLconf для краткости). Шаблон проекта по умолчанию определяет URLconf в `<myproject>/urls.py` .

Ваш URLconf должен быть модулем python, который определяет атрибут с именем `urlpatterns` , который является списком `django.conf.urls.url()` . Каждый экземпляр `url()` должен, как минимум, определять регулярное выражение (регулярное выражение) для сопоставления с URL-адресом и цель, которая является либо функцией просмотра, либо другим URL-интерфейсом. Если шаблон URL нацелен на функцию просмотра, рекомендуется дать ему имя, чтобы легко ссылаться на шаблон позже.

Давайте рассмотрим базовый пример:

```
# In <myproject>/urls.py

from django.conf.urls import url

from myapp.views import home, about, blog_detail

urlpatterns = [
    url(r'^$', home, name='home'),
    url(r'^about/$', about, name='about'),
    url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),
]
```

Этот URLconf определяет три шаблона URL-адресов: все таргетинг на представление: `home` страница, информация `about` `blog-detail` и `blog-detail` .

- `url(r'^$', home, name='home'),`

Регулярное выражение содержит стартовый якорь '^', за которым следует конечный якорь '$'. Этот шаблон будет соответствовать запросам, где путь URL-адреса является пустой строкой, и `myapp.views` их в представление `home` определенное в `myapp.views` .

- `url(r'^about/$', about, name='about'),`

Это регулярное выражение содержит стартовый якорь, за которым следует буквальная строка `about/` и конечный якорь. Это будет соответствовать URL `/about/` и маршрут к `about`

представлении. Поскольку каждый непустой URL-адрес начинается с `/` , Django удобно разрезает первую косую черту для вас.

- `url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),`

Это регулярное выражение немного сложнее. Он определяет стартовый якорь и буквенный строковый `blog/` , как и предыдущий шаблон. Следующая часть `(?P<id>\d+)` называется группой захвата. Группа захвата, как и его название, фиксирует часть строки, а Django передает захваченную строку в качестве аргумента функции просмотра.

Синтаксис группы захвата - это `(?P<name>pattern)` . `name` определяет имя группы, которое также является именем, которое Django использует для передачи аргумента в представление. Шаблон определяет, какие символы соответствуют группе.

В этом случае имя является `id` , поэтому функция `blog_detail` должна принимать параметр с именем `id` . Шаблон равен `\d+` . `\d` означает, что шаблон соответствует только номерам символов. `+` означает, что шаблон должен соответствовать одному или нескольким символам.

Некоторые общие закономерности:

Шаблон	Используется для	Матчи
`\d+`	Я бы	Один или несколько числовых символов
`[\w-]+`	слизень	Один или несколько буквенно-цифровых символов, подчеркивание или тире
`[0-9]{4}`	год (длинный)	Четыре числа, от нуля до девяти
`[0-9]{2}`	год (короткий) месяц день месяца	Два числа, от нуля до девяти
`[^/]+`	сегмент пути	Все, кроме косой черты

Группа захвата в шаблоне `blog-detail` сопровождается литералом `/` и конечным якорем.

Допустимые URL-адреса:

- `/blog/1/ # passes id='1'`
- `/blog/42/ # passes id='42'`

Недействительными URL-адресами являются, например:

- `/blog/a/ # 'a' does not match '\d'`
- `/blog// # no characters in the capturing group does not match '+'`

Django обрабатывает каждый шаблон URL в том же порядке, который определен в `urlpatterns` . Это важно, если несколько шаблонов могут совпадать с одним и тем же URL. Например:

```
urlpatterns = [
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
]
```

В приведенном выше URLconf второй шаблон недоступен. Шаблон будет соответствовать URL `/blog/overview/` , но вместо вызова вида `blog_overview` URL-адрес сначала будет соответствовать шаблону `blog-detail` и вызовет представление `blog_detail` с аргументом `slug='overview'` .

Чтобы убедиться, что URL `/blog/overview/` перенаправлен в представление `blog_overview` , шаблон должен быть помещен над шаблоном `blog-detail` :

```
urlpatterns = [
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
]
```

Задайте пространство имен URL для многоразового приложения (Django 1.9+)

Настройте URLconf вашего приложения для автоматического использования пространства имен URL, установив атрибут `app_name` :

```
# In <myapp>/urls.py
from django.conf.urls import url

from .views import overview

app_name = 'myapp'
urlpatterns = [
    url(r'^$', overview, name='overview'),
]
```

Это установит пространство имен приложений на `'myapp'` когда оно будет включено в корневой URLconf>. Пользователю вашего многоразового приложения не нужно выполнять какую-либо конфигурацию, кроме как включать ваши URL-адреса:

```
# In <myproject>/urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^myapp/', include('myapp.urls')),
]
```

Теперь ваше многоразовое приложение может отменить URL-адреса, используя

пространство имен приложений:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
```

Корневой URLconf все еще может устанавливать пространство имен экземпляров с параметром `namespace` :

```
# In <myproject>/urls.py
urlpatterns = [
    url(r'^myapp/', include('myapp.urls', namespace='mynamespace')),
]
```

И пространство имен приложений, и пространство имен экземпляров могут использоваться для изменения URL-адресов:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
>>> reverse('mynamespace:overview')
'/myapp/overview/'
```

Пространство имен экземпляров по умолчанию используется для пространства имен приложений, если оно явно не задано.

Прочитайте Маршрутизация URL онлайн: https://riptutorial.com/ru/django/topic/3299/маршрутизация-url

глава 29: Миграции

параметры

Команда `django-admin`	подробности
`makemigrations <my_app>`	Создание миграции для `my_app`
`makemigrations`	Создание миграций для всех приложений
`makemigrations --merge`	Устранение конфликтов миграции для всех приложений
`makemigrations --merge <my_app>`	`my_app` конфликтов миграции для `my_app`
`makemigrations --name <migration_name> <my_app>`	Сформировать миграции для `my_app` с именем `migration_name`
`migrate <my_app>`	Применить ожидающие миграции `my_app` в базу данных
`migrate`	Применить все ожидающие миграции в базу данных
`migrate <my_app> <migration_name>`	Применить или исключить до `migration_name`
`migrate <my_app> zero`	Не использовать все миграции в `my_app`
`sqlmigrate <my_app> <migration_name>`	Распечатывает SQL для указанной миграции
`showmigrations`	Показывает все миграции для всех приложений
`showmigrations <my_app>`	Показывает все миграции в `my_app`

Examples

Работа с миграциями

Django использует миграции для распространения изменений, которые вы делаете на свои модели, в свою базу данных. В большинстве случаев django может генерировать их для вас.

Чтобы создать миграцию, выполните:

```
$ django-admin makemigrations <app_name>
```

Это создаст файл `migration` подмодуле `migration app_name` . Первая миграция будет называться `0001_initial.py` , другая начнется с `0002_` , затем `0003` , ...

Если вы опустите `<app_name>` это создаст миграцию для всех ваших `INSTALLED_APPS` .

Чтобы распространять миграцию в вашу базу данных, запустите:

```
$ django-admin migrate <app_name>
```

Чтобы показать все ваши миграции, запустите:

```
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [X] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

- `[X]` означает, что миграция была передана в вашу базу данных
- `[ ]` означает, что миграция не была распространена в вашей базе данных. Используйте `django-admin migrate` для распространения

Вы также вызываете и возвращаете миграцию, это можно сделать, передав имя `migrate command` . Учитывая приведенный выше список миграций (показано с помощью `django-admin showmigrations`):

```
$ django-admin migrate app_name 0002  # Roll back to migration 0002
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [ ] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

Ручная миграция

Иногда миграций, создаваемых Django, недостаточно. Это особенно актуально, если вы хотите выполнить **миграцию данных** .

Например, давайте иметь такую модель:

```
class Article(models.Model):
    title = models.CharField(max_length=70)
```

У этой модели уже есть существующие данные, и теперь вы хотите добавить `SlugField` :

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

```
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70)
```

Вы создали миграцию для добавления поля, но теперь вы хотите установить пул для всей существующей статьи в соответствии с их `title` .

Конечно, вы можете просто сделать что-то подобное в терминале:

```
$ django-admin shell
>>> from my_app.models import Article
>>> from django.utils.text import slugify
>>> for article in Article.objects.all():
...     article.slug = slugify(article.title)
...     article.save()
...
>>>
```

Но вам придется делать это во всех своих средах (например, на рабочем столе вашего офиса, на вашем ноутбуке ...), всем вашим коллегам также придется это делать, и вам придется подумать об этом при постановке и при нажатии жить.

Чтобы сделать это раз и навсегда, мы сделаем это в процессе миграции. Сначала создайте пустую миграцию:

```
$ django-admin makemigrations --empty app_name
```

Это создаст пустой файл миграции. Откройте его, он содержит базовый скелет. Предположим, что ваша предыдущая миграция была названа `0023_article_slug` и она называется `0024_auto_20160719_1734` . Вот что мы напишем в нашем файле миграции:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.9.7 on 2016-07-19 15:34
from __future__ import unicode_literals

from django.db import migrations
from django.utils.text import slugify


def gen_slug(apps, schema_editor):
    # We can't import the Article model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Article = apps.get_model('app_name', 'Article')
    for row in Article.objects.all():
        row.slug = slugify(row.name)
        row.save()


class Migration(migrations.Migration):

    dependencies = [
        ('hosting', '0023_article_slug'),
    ]
```

```
    operations = [
        migrations.RunPython(gen_slug, reverse_code=migrations.RunPython.noop),
        # We set `reverse_code` to `noop` because we cannot revert the migration
        # to get it back in the previous state.
        # If `reverse_code` is not given, the migration will not be reversible,
        # which is not the behaviour we expect here.
    ]
```

Поддельные миграции

Когда миграция выполняется, Django сохраняет имя миграции в таблице django_migrations.

Создание и подделка начальных миграций для существующей схемы

Если ваше приложение уже имеет модели и таблицы базы данных и не имеет миграций. Сначала создайте начальные миграции для своего приложения.

```
python manage.py makemigrations your_app_label
```

Теперь поддельные начальные миграции применимы

```
python manage.py migrate --fake-initial
```

Поддельные все миграции во всех приложениях

```
python manage.py migrate --fake
```

Поддельные миграции приложений

```
python manage.py migrate --fake core
```

Поддельный файл с одной миграцией

```
python manage.py migrate myapp migration_name
```

Пользовательские имена для файлов миграции

Используйте параметр `makemigrations --name <your_migration_name>` чтобы разрешить именовать миграцию (-и) вместо использования сгенерированного имени.

```
python manage.py makemigrations --name <your_migration_name> <app_name>
```

Устранение конфликтов миграции

Вступление

Иногда конфликты конфликтуют, в результате чего миграция не выполняется. Это может произойти во множестве scenerio, однако это может происходить на регулярной основе при разработке одного приложения с командой.

Общие конфликты миграции происходят при использовании контроля источника, особенно когда используется метод «функция-ветвь». Для этого сценария мы будем использовать модель `Reporter` с `name` и `address` атрибутов.

Два разработчика на этом этапе собираются разработать функцию, поэтому они оба получат эту исходную копию модели `Reporter` . Разработчик A добавляет `age` который приводит к файлу `0002_reporter_age.py` . Разработчик B добавляет поле `bank_account` которое `0002_reporter_bank_account` в `0002_reporter_bank_account` . Как только эти разработчики объединит свой код и попытаются перенести миграции, возник конфликт миграции.

Этот конфликт возникает, потому что эти миграции изменяют одну и ту же модель, `Reporter` . Кроме того, новые файлы начинаются с 0002.

Слияние миграций

Есть несколько способов сделать это. В рекомендуемом порядке:

1. Самое простое решение для этого - запустить команду makemigrations с флагом - merge.

   ```
   python manage.py makemigrations --merge <my_app>
   ```

   Это создаст новую миграцию, разрешающую предыдущий конфликт.

2. Когда этот дополнительный файл не приветствуется в среде разработки по личным причинам, опция заключается в удалении конфликтующих миграций. Затем новая миграция может быть выполнена с помощью обычной команды `makemigrations` . Когда пользовательские миграции записываются, например `migrations.RunPython` , необходимо учитывать этот метод.

Измените CharField на ForeignKey

Прежде всего, давайте предположим, что это ваша первоначальная модель внутри приложения, называемого `discography` :

```
from django.db import models

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

Теперь вы понимаете, что вместо этого вы хотите использовать ForeignKey для

исполнителя. Это несколько сложный процесс, который необходимо выполнить в несколько этапов.

Шаг 1, добавьте новое поле для ForeignKey, убедившись, что он отмечен как null (обратите внимание, что теперь мы подключаем модель):

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
    artist_link = models.ForeignKey('Artist', null=True)

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

... и создать миграцию для этого изменения.

```
./manage.py makemigrations discography
```

Шаг 2, заполните новое поле. Для этого вам нужно создать пустую миграцию.

```
./manage.py makemigrations --empty --name transfer_artists discography
```

Когда у вас будет эта пустая миграция, вы хотите добавить к ней одну операцию `RunPython` , чтобы связать ваши записи. В этом случае он может выглядеть примерно так:

```
def link_artists(apps, schema_editor):
    Album = apps.get_model('discography', 'Album')
    Artist = apps.get_model('discography', 'Artist')
    for album in Album.objects.all():
        artist, created = Artist.objects.get_or_create(name=album.artist)
        album.artist_link = artist
        album.save()
```

Теперь, когда ваши данные будут перенесены в новое поле, вы действительно можете сделать и оставить все как есть, используя новое поле `artist_link` для всего. Или, если вы хотите немного очистить, вы хотите создать еще две миграции.

Для первой миграции вы хотите удалить свое исходное поле, `artist` . Для вашей второй миграции переименуйте новое поле `artist_link` в `artist` .

Это делается несколькими шагами, чтобы гарантировать, что Django правильно распознает операции.

Прочитайте Миграции онлайн: https://riptutorial.com/ru/django/topic/1200/миграции

глава 30: модели

Вступление

В базовом случае модель представляет собой класс Python, который сопоставляется с одной таблицей базы данных. Атрибуты карты классов для столбцов в таблице и экземпляр класса представляют строку в таблице базы данных. Модели наследуют от `django.db.models.Model` который предоставляет богатый API для добавления и фильтрации результатов из базы данных.

Создайте свою первую модель

Examples

Создание вашей первой модели

Модели обычно определяются в файле `models.py` в подкаталоге вашего приложения. Класс `Model` модуля `django.db.models` является хорошим стартовым классом для расширения ваших моделей. Например:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey('Author', on_delete=models.CASCADE,
related_name='authored_books')
    publish_date = models.DateField(null=True, blank=True)

    def __str__(self): # __unicode__ in python 2.*
        return self.title
```

Каждый атрибут в модели представляет собой столбец в базе данных.

- `title` - текст с максимальной длиной 100 символов
- `author` - `ForeignKey` который представляет собой отношение к другой модели / таблице, в данном случае `Author` (используется только для примера). `on_delete` сообщает базе данных, что делать с объектом, если связанный объект (`Author`) будет удален. (Следует отметить, что поскольку django 1.9 `on_delete` может использоваться как второй позиционный аргумент. В django 2 это **обязательный** аргумент, и рекомендуется сразу же рассматривать его как таковое немедленно. В более старых версиях он будет по умолчанию для `CASCADE` .)
- `publish_date` хранит дату. Оба `null` и `blank` равны `True` чтобы указать, что это не обязательное поле (т. Е. Вы можете добавить его позже или оставить пустым).

Наряду с атрибутами мы определяем метод `__str__` это возвращает заголовок книги,

который будет использоваться в качестве его `string` представления, где это необходимо, а не по умолчанию.

Применение изменений в базе данных (Миграции)

После создания новой модели или изменения существующих моделей вам необходимо будет создать миграцию для ваших изменений, а затем применить миграции к указанной базе данных. Это можно сделать, используя встроенную систему миграции Django. Использование утилиты `manage.py` в корневом каталоге проекта:

```
python manage.py makemigrations <appname>
```

Вышеприведенная команда создаст сценарии миграции, которые необходимы в подкаталоге `migrations` вашего приложения. Если вы опустите параметр `<appname>` , все приложения, определенные в аргументе `INSTALLED_APPS` параметра `settings.py` будут обработаны. Если вы сочтете это необходимым, вы можете редактировать миграцию.

Вы можете проверить, какие миграции требуются без фактического создания миграции, используйте параметр -dry-run, например:

```
python manage.py makemigrations --dry-run
```

Чтобы применить миграции:

```
python manage.py migrate <appname>
```

Вышеупомянутая команда выполнит сценарии миграции, сгенерированные на первом этапе, и физически обновит базу данных.

Если изменена модель существующей базы данных, для внесения необходимых изменений необходима следующая команда.

```
python manage.py migrate --run-syncdb
```

Django создаст таблицу с именем `<appname>_<classname>` по умолчанию. Иногда вы не хотите его использовать. Если вы хотите изменить имя по умолчанию, вы можете объявить имя таблицы, установив `db_table` в класс `Meta` :

```
from django.db import models

class YourModel(models.Model):
    parms = models.CharField()
    class Meta:
        db_table = "custom_table_name"
```

Если вы хотите увидеть, какой код SQL будет выполняться с помощью определенной

миграции, просто запустите эту команду:

```
python manage.py sqlmigrate <app_label> <migration_number>
```

Джанго> 1.10
Новая опция `makemigrations --check` делает `makemigrations --check` команды с ненулевым статусом при обнаружении изменений модели без миграции.

См. « Миграции» для получения более подробной информации о миграции.

Создание модели с отношениями

Отношения «много-к-одному»

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

#Book has a foreignkey (many to one) relationship with author
class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    publish_date = models.DateField()
```

Самый общий вариант. Может использоваться везде, где вы хотели бы представить отношения

Отношения «многие ко многим»

```
class Topping(models.Model):
    name = models.CharField(max_length=50)

# One pizza can have many toppings and same topping can be on many pizzas
class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)
```

Внутренне это представлено через другую таблицу. И `ManyToManyField` следует помещать в модели, которые будут редактироваться в форме. Например: `Appointment` будет иметь `ManyToManyField` под названием `Customer` , `Pizza` имеет `Toppings` и так далее.

Использование отношений «многие-ко-многим» с использованием сквозных классов

```
class Service(models.Model):
     name = models.CharField(max_length=35)

class Client(models.Model):
    name = models.CharField(max_length=35)
    age = models.IntegerField()
    services = models.ManyToManyField(Service, through='Subscription')
```

```
class Subscription(models.Model):
     client = models.ForeignKey(Client)
     service = models.ForeignKey(Service)
     subscription_type = models.CharField(max_length=1, choices=SUBSCRIPTION_TYPES)
     created_at = models.DateTimeField(default=timezone.now)
```

Таким образом, мы можем сохранить больше метаданных о взаимосвязи между двумя объектами. Как видно, клиент может быть подписан на несколько сервисов несколькими типами подписки. Единственное отличие в этом случае заключается в том, что для добавления новых экземпляров в отношение M2M нельзя использовать метод ярлыков `pizza.toppings.add(topping)` , вместо этого должен быть создан новый объект класса *through* , `Subscription.objects.create(client=client, service=service, subscription_type='p')`

> В других языках `through tables` также известны как `JoinColumn` , `Intersection table` или `mapping table`

Индивидуальные отношения

```
class Employee(models.Model):
   name = models.CharField(max_length=50)
   age = models.IntegerField()
   spouse = models.OneToOneField(Spouse)

class Spouse(models.Model):
   name = models.CharField(max_length=50)
```

Используйте эти поля, когда у вас будет только соотношение композиций между двумя моделями.

Основные запросы Django DB

Django ORM - это мощная абстракция, которая позволяет хранить и извлекать данные из базы данных без написания запросов sql.

Предположим следующие модели:

```
class Author(models.Model):
   name = models.CharField(max_length=50)

class Book(models.Model):
   name = models.CharField(max_length=50)
   author = models.ForeignKey(Author)
```

Предполагая, что вы добавили вышеуказанный код в приложение django и запустите команду `migrate` (чтобы ваша база данных была создана). Запустите оболочку Django на

```
python manage.py shell
```

Это запускает стандартную оболочку python, но с импортированными соответствующими

библиотеками Django, так что вы можете напрямую сосредоточиться на важных частях.

Начните с импорта только что определенных моделей (я предполагаю, что это делается в файле `models.py`)

```
from .models import Book, Author
```

Запустите свой первый запрос выбора:

```
>>> Author.objects.all()
[]
>>> Book.objects.all()
[]
```

Позволяет создать объект автора и книги:

```
>>> hawking = Author(name="Stephen hawking")
>>> hawking.save()
>>> history_of_time = Book(name="history of time", author=hawking)
>>> history_of_time.save()
```

или использовать функцию create для создания объектов модели и сохранения в одной строке кода

```
>>> wings_of_fire = Book.objects.create(name="Wings of Fire", author="APJ Abdul Kalam")
```

Теперь запустите запрос

```
>>> Book.objects.all()
[<Book: Book object>]
>>> book = Book.objects.first() #getting the first book object
>>> book.name
u'history of time'
```

Давайте добавим предложение where к нашему запросу select

```
>>> Book.objects.filter(name='nothing')
[]
>>> Author.objects.filter(name__startswith='Ste')
[<Author: Author object>]
```

Чтобы получить подробную информацию об авторе данной книги

```
>>> book = Book.objects.first() #getting the first book object
>>> book.author.name # lookup on related model
u'Stephen hawking'
```

Чтобы получить все книги, опубликованные Стивеном Хокином (Lookup book от его автора)

```
>>> hawking.book_set.all()
[<Book: Book object>]
```

`_set` - это обозначение, используемое для «Обратного поиска», т. е. когда поле поиска находится в модели книги, мы можем использовать `book_set` для объекта автора, чтобы получить все его книги.

Основная неуправляемая таблица.

В какой-то момент использования Django вы можете захотеть взаимодействовать с уже созданными таблицами или с представлениями базы данных. В этих случаях вы не хотите, чтобы Django управлял таблицами через свои миграции. Чтобы установить это, вам нужно добавить только одну переменную в `Meta` класс вашей модели: `managed = False` .

Ниже приведен пример того, как создать неуправляемую модель для взаимодействия с представлением базы данных:

```
class Dummy(models.Model):
    something = models.IntegerField()

    class Meta:
       managed = False
```

Это может быть сопоставлено с представлением, определенным в SQL, следующим образом.

```
CREATE VIEW myapp_dummy AS
SELECT id, something FROM complicated_table
WHERE some_complicated_condition = True
```

Когда вы создадите эту модель, вы можете использовать ее, как и любую другую модель:

```
>>> Dummy.objects.all()
[<Dummy: Dummy object>, <Dummy: Dummy object>, <Dummy: Dummy object>]
>>> Dummy.objects.filter(something=42)
[<Dummy: Dummy object>]
```

Расширенные модели

Модель может предоставить гораздо больше информации, чем просто данные об объекте. Давайте посмотрим пример и разделим его на то, что он полезен для:

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    slug = models.SlugField()
```

```
    title = models.CharField(max_length=128)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('library:book', kwargs={'pk':self.pk})

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['publish_date', 'title']
```

Автоматический первичный ключ

Вы можете заметить использование `self.pk` в методе `get_absolute_url` . Поле `pk` является псевдонимом первичного ключа модели. Кроме того, django автоматически добавит первичный ключ, если он отсутствует. Это еще одна вещь, о которой нужно беспокоиться, и дать вам возможность установить внешний ключ для любых моделей и получить их легко.

Абсолютный URL-адрес

Первая функция, которая определена, - `get_absolute_url` . Таким образом, если у вас есть книга, вы можете получить ссылку на нее без использования тэга url, разрешения, атрибута и тому подобного. Просто позвоните в `book.get_absolute_url` и вы получите правильную ссылку. В качестве бонуса ваш объект в администраторе django получит кнопку «просмотр на сайте».

Строковое представление

`__str__` метод `__str__` чтобы использовать объект, когда вам нужно его отобразить. Например, с помощью предыдущего метода добавление ссылки в книгу в шаблон так же просто, как `<a href="{{ book.get_absolute_url }}">{{ book }}</a>` . Прямо к сути. Этот метод также контролирует то, что отображается в раскрывающемся списке admin, например, для внешнего ключа.

Декоратор класса позволяет вам определить метод один раз для `__str__` и `__unicode__` на python 2, не вызывая проблем на python 3. Если вы ожидаете, что ваше приложение будет работать на обеих версиях, это путь.

Поле слизняков

Поле slug похоже на поле char, но принимает меньше символов. По умолчанию используются только буквы, цифры, символы подчеркивания или дефисы. Это полезно, если вы хотите идентифицировать объект, используя хорошее представление, например, в

URL-адресе.

Класс Meta

Класс `Meta` позволяет нам определять намного больше информации по всему набору предметов. Здесь задается только порядок по умолчанию. Это полезно, например, для объекта ListView. Для сортировки требуется идеально короткий список полей. Здесь книга будет отсортирована сначала по дате публикации, а затем по названию, если дата будет одинаковой.

Другими атрибутами `verbose_name` являются `verbose_name` и `verbose_name_plural` . По умолчанию они генерируются из имени модели и должны быть точными. Но множественная форма наивна, просто добавляет 's' к единственному, поэтому вы можете явно указать его в некотором случае.

Вычисляемые значения

После того, как объект модели был извлечен, он становится полностью реализованным экземпляром класса. Таким образом, любые дополнительные методы могут быть доступны в формах и сериализаторах (например, в Django Rest Framework).

Использование свойств python - это элегантный способ представления дополнительных значений, которые не хранятся в базе данных из-за различных обстоятельств.

```
def expire():
    return timezone.now() + timezone.timedelta(days=7)

class Coupon(models.Model):
    expiration_date = models.DateField(default=expire)

    @property
    def is_expired(self):
        return timezone.now() > self.expiration_date
```

Хотя в большинстве случаев вы можете дополнить данные аннотациями на своих запросах, вычисленные значения в качестве свойств модели идеальны для вычислений, которые невозможно оценить просто в рамках запроса.

Кроме того, свойства, поскольку они объявлены в классе python, а не как часть схемы, недоступны для запроса.

Добавление строкового представления модели

Чтобы создать удобочитаемое представление объекта модели, вам необходимо реализовать метод `Model.__str__()` (или `Model.__unicode__()` на python2). Этот метод будет вызываться всякий раз, когда вы вызываете `str()` в экземпляре вашей модели (включая,

например, когда модель используется в шаблоне). Вот пример:

1. Создайте модель книги.

```
# your_app/models.py

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
```

2. Создайте экземпляр модели и сохраните ее в базе данных:

```
>>> himu_book = Book(name='Himu Mama', author='Humayun Ahmed')
>>> himu_book.save()
```

3. Выполните `print()` в экземпляре:

```
>>> print(himu_book)
<Book: Book object>
```

<Book: Объект Book> , выходной по умолчанию, не помогает нам. Чтобы исправить это, добавим метод `__str__` .

```
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)

    def __str__(self):
        return '{} by {}'.format(self.name, self.author)
```

Обратите внимание, что `python_2_unicode_compatible` decorator необходим, только если вы хотите, чтобы ваш код был совместим с python 2. Этот декоратор копирует метод `__str__` для создания метода `__unicode__` . Импортируйте его из `django.utils.encoding` .

Теперь, если мы снова назовем функцию печати экземпляром книги:

```
>>> print(himu_book)
Himu Mama by Humayun Ahmed
```

Намного лучше!

Строковое представление также используется, когда модель используется в поля `ModelForm` для `ForeignKeyField` и `ManyToManyField` .

Модельные смеси

В тех же случаях разные модели могут иметь одинаковые поля и одинаковые процедуры в жизненном цикле продукта. Чтобы справиться с этими сходствами, не используя наследование повторения кода, можно было бы использовать. Вместо наследования целого класса шаблон проектирования **mixin** предлагает нам наследовать (*или некоторые из них включать*) некоторые методы и атрибуты. Давайте посмотрим пример:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    sender_name = models.CharField(max_length=128)
    sender_address = models.CharField(max_length=255)
    receiver_name = models.CharField(max_length=128)
    receiver_address = models.CharField(max_length=255)
    post_datetime = models.DateTimeField(auto_now_add=True)
    delivery_datetime = models.DateTimeField(null=True)
    notes = models.TextField(max_length=500)

class Envelope(PostableMixin):
    ENVELOPE_COMMERCIAL = 1
    ENVELOPE_BOOKLET = 2
    ENVELOPE_CATALOG = 3

    ENVELOPE_TYPES = (
        (ENVELOPE_COMMERCIAL, 'Commercial'),
        (ENVELOPE_BOOKLET, 'Booklet'),
        (ENVELOPE_CATALOG, 'Catalog'),
    )

    envelope_type = models.PositiveSmallIntegerField(choices=ENVELOPE_TYPES)

class Package(PostableMixin):
    weight = models.DecimalField(max_digits=6, decimal_places=2)
    width = models.DecimalField(max_digits=5, decimal_places=2)
    height = models.DecimalField(max_digits=5, decimal_places=2)
    depth = models.DecimalField(max_digits=5, decimal_places=2)
```

Чтобы превратить модель в абстрактный класс, вам нужно будет указать `abstract=True` в своем внутреннем `Meta` классе. Django не создает таблицы для абстрактных моделей в базе данных. Однако для моделей `Envelope` и `Package` в базе данных будут созданы соответствующие таблицы.

Кроме того, в некоторых моделях потребуются некоторые модельные методы. Таким образом, эти методы могут быть добавлены в mixins для предотвращения повторения кода. Например, если мы создадим метод установки даты доставки в `PostableMixin` он будет доступен от обоих его дочерних элементов:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    ...
    ...
```

```
    def set_delivery_datetime(self, dt=None):
        if dt is None:
            from django.utils.timezone import now
            dt = now()

        self.delivery_datetime = dt
        self.save()
```

Этот метод можно использовать для детей следующим образом:

```
>> envelope = Envelope.objects.get(pk=1)
>> envelope.set_delivery_datetime()

>> pack = Package.objects.get(pk=1)
>> pack.set_delivery_datetime()
```

Первичный ключ UUID

Модель по умолчанию будет использовать первичный ключ auto incrementing (integer). Это даст вам последовательность клавиш 1, 2, 3.

Различные типы первичных ключей могут быть установлены на модели с небольшими изменениями модели.

UUID является универсально уникальным идентификатором, это 32-символьный случайный идентификатор, который может использоваться как идентификатор. Это хороший вариант использования, когда вы не хотите, чтобы в вашей базе данных были назначены последовательные идентификаторы. При использовании в PostgreSQL это хранится в виде данных uuid, иначе в char (32).

```
import uuid
from django.db import models

class ModelUsingUUID(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
```

Сгенерированный ключ будет в формате `7778c552-73fc-4bc4-8bf9-5a2f6f7b7f47`

наследование

Наследование между моделями можно сделать двумя способами:

- общий абстрактный класс (см. пример «Модельные миксины»)
- общая модель с несколькими таблицами

Наследование множественных таблиц создаст одну таблицу для общих полей и по одному на пример для дочерней модели:

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

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

создаст 2 таблицы для `Place` и один для `Restaurant` со скрытым `OneToOne` полем `Place` для общих полей.

обратите внимание, что при каждом приеме объекта «Ресторан» потребуется дополнительный запрос к таблицам мест.

Прочитайте модели онлайн: https://riptutorial.com/ru/django/topic/888/модели

глава 31: Модельные агрегации

Вступление

Агрегации - это методы, позволяющие выполнять операции над (индивидуальными и / или группами) строк объектов, полученных из Модели.

Examples

Среднее, Минимальное, Максимальное, Сумма от Queryset

```
class Product(models.Model):
    name = models.CharField(max_length=20)
    price = models.FloatField()
```

Получить среднюю цену всех продуктов:

```
>>> from django.db.models import Avg, Max, Min, Sum
>>> Product.objects.all().aggregate(Avg('price'))
# {'price__avg': 124.0}
```

Чтобы получить минимальную цену для всех продуктов:

```
>>> Product.objects.all().aggregate(Min('price'))
# {'price__min': 9}
```

Получить максимальную цену для всех продуктов:

```
>>> Product.objects.all().aggregate(Max('price'))
# {'price__max':599 }
```

Чтобы получить СУММУ всех продуктов:

```
>>> Product.objects.all().aggregate(Sum('price'))
# {'price__sum':92456 }
```

Подсчитайте количество внешних связей

```
class Category(models.Model):
    name = models.CharField(max_length=20)


class Product(models.Model):
    name = models.CharField(max_length=64)
    category = models.ForeignKey(Category, on_delete=models.PROTECT)
```

Чтобы получить число продуктов для каждой категории:

```
>>> categories = Category.objects.annotate(Count('product'))
```

Это добавляет атрибут `<field_name>__count` для каждого возвращаемого экземпляра:

```
>>> categories.values_list('name', 'product__count')
[('Clothing', 42), ('Footwear', 12), ...]
```

Вы можете указать собственное имя для своего атрибута, используя аргумент ключевого слова:

```
>>> categories = Category.objects.annotate(num_products=Count('product'))
```

Вы можете использовать аннотированное поле в запросах:

```
>>> categories.order_by('num_products')
[<Category: Footwear>, <Category: Clothing>]

>>> categories.filter(num_products__gt=20)
[<Category: Clothing>]
```

GROUB BY ... COUNT / SUM эквивалент Django ORM

Мы можем выполнить `GROUP BY ... COUNT` или через `GROUP BY ... SUM` SQL эквивалентных запросов на Django ORM, с использованием `annotate()` , `values()` , `order_by()` и `django.db.models` «S `Count` и `Sum` методы с уважением:

Пусть наша модель будет:

```
   class Books(models.Model):
       title  = models.CharField()
       author = models.CharField()
       price = models.FloatField()
```

`GROUP BY ... COUNT` :

- Предположим, что мы хотим подсчитать, сколько книжных объектов на отдельный автор существует в нашей таблице « `Books` :

  ```
  result = Books.objects.values('author')
                        .order_by('author')
                        .annotate(count=Count('author'))
  ```

- Теперь `result` содержит набор запросов с двумя столбцами: `author` и `count` :

  ```
    author    | count
  ------------|-------
  ```

```
 OneAuthor  |   5
OtherAuthor |   2
   ...      |  ...
```

GROUB BY ... SUM :

- Предположим, что мы хотим суммировать цену всех книг для отдельного автора, которые существуют в нашей таблице « `Books` :

```
result = Books.objects.values('author')
                      .order_by('author')
                      .annotate(total_price=Sum('price'))
```

- Теперь `result` содержит набор запросов с двумя столбцами: `author` и `total_price` :

```
  author    | total_price
------------|-------------
 OneAuthor  |    100.35
OtherAuthor |     50.00
   ...      |      ...
```

Прочитайте Модельные агрегации онлайн: https://riptutorial.com/ru/django/topic/3775/модельные-агрегации

глава 32: Настройка базы данных

Examples

MySQL / MariaDB

Django поддерживает MySQL 5.5 и выше.

Убедитесь, что установлены некоторые пакеты:

```
$ sudo apt-get install mysql-server libmysqlclient-dev
$ sudo apt-get install python-dev python-pip               # for python 2
$ sudo apt-get install python3-dev python3-pip             # for python 3
```

Как и один из драйверов Python MySQL (`mysqlclient` соответствует рекомендуемому выбору для Django):

```
$ pip install mysqlclient    # python 2 and 3
$ pip install MySQL-python   # python 2
$ pip install pymysql        # python 2 and 3
```

> Кодирование базы данных не может быть установлено Django, но должно быть настроено на уровне базы данных. Найдите `default-character-set` по `default-character-set` в my.cnf (или `/etc/mysql/mariadb.conf/*.cnf`) и установите кодировку:

```
[mysql]
#default-character-set = latin1    #default on some systems.
#default-character-set = utf8mb4   #default on some systems.
default-character-set = utf8

...
[mysqld]
#character-set-server  = utf8mb4
#collation-server      = utf8mb4_general_ci
character-set-server  = utf8
collation-server      = utf8_general_ci
```

Конфигурация базы данных для MySQL или MariaDB

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DB_NAME',
        'USER': 'DB_USER',
        'PASSWORD': 'DB_PASSWORD',
        'HOST': 'localhost',   # Or an IP Address that your database is hosted on
        'PORT': '3306',
        #optional:
```

```
        'OPTIONS': {
            'charset' : 'utf8',
            'use_unicode' : True,
             'init_command': 'SET '
                'storage_engine=INNODB,'
                'character_set_connection=utf8,'
                'collation_connection=utf8_bin'
                #'sql_mode=STRICT_TRANS_TABLES,'    # see note below
                #'SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED',
        },
        'TEST_CHARSET': 'utf8',
        'TEST_COLLATION': 'utf8_general_ci',
    }
}
```

Если вы используете соединитель MySQL Oracle, ваша линия `ENGINE` должна выглядеть так:

```
'ENGINE': 'mysql.connector.django',
```

Когда вы создаете базу данных, убедитесь, что для указания кодировки и сопоставления:

```
CREATE DATABASE mydatabase CHARACTER SET utf8 COLLATE utf8_bin
```

> Начиная с MySQL 5.7 и последующих **обновлений** MySQL 5.6 значение по умолчанию для параметра sql_mode содержит **STRICT_TRANS_TABLES** . Этот параметр увеличивает предупреждения при ошибках, когда данные усекаются при вставке. Django настоятельно рекомендует активировать *строгий режим* для MySQL, чтобы предотвратить потерю данных (STRICT_TRANS_TABLES или STRICT_ALL_TABLES). Чтобы включить добавление в */etc/my.cnf* `sql-mode = STRICT_TRANS_TABLES`

PostgreSQL

Убедитесь, что установлены некоторые пакеты:

```
sudo apt-get install libpq-dev
pip install psycopg2
```

Настройки базы данных для PostgreSQL:

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'myprojectDB',
        'USER': 'myprojectuser',
        'PASSWORD': 'password',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

В старых версиях вы также можете использовать псевдоним `django.db.backends.postgresql_psycopg2` .

При использовании Postresql у вас будет доступ к некоторым дополнительным функциям:

Modelfields:

```
ArrayField         # A field for storing lists of data.
HStoreField        # A field for storing mappings of strings to strings.
JSONField          # A field for storing JSON encoded data.
IntegerRangeField  # Stores a range of integers
BigIntegerRangeField # Stores a big range of integers
FloatRangeField    # Stores a range of floating point values.
DateTimeRangeField # Stores a range of timestamps
```

SQLite

sqlite по умолчанию для Django. *Он не должен использоваться в производстве, поскольку он обычно медленный.*

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db/development.sqlite3',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    },
}
```

арматура

Светильники являются исходными данными для базы данных. Самый простой способ, когда у вас есть уже существующие данные, это использовать команду `dumpdata`

```
 ./manage.py dumpdata > databasedump.json                # full database
 ./manage.py dumpdata myapp > databasedump.json          # only 1 app
 ./manage.py dumpdata myapp.mymodel > databasedump.json  # only 1 model (table)
```

Это создаст json-файл, который можно импортировать снова, используя

```
./manage.py loaddata databasedump.json
```

При использовании `loadddata` без указания файла, Django будет искать `fixtures` папки в вашем приложении или в списке каталогов , предусмотренных в `FIXTURE_DIRS` в настройках, и использовать его содержимое вместо этого.

```
/myapp
   /fixtures
        myfixtures.json
        morefixtures.xml
```

Возможные форматы файлов: `JSON, XML or YAML`

Светильники Пример JSON:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

Светильники Пример YAML:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

Пример XML XML:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="1" model="myapp.person">
        <field type="CharField" name="first_name">John</field>
        <field type="CharField" name="last_name">Lennon</field>
    </object>
    <object pk="2" model="myapp.person">
        <field type="CharField" name="first_name">Paul</field>
        <field type="CharField" name="last_name">McCartney</field>
    </object>
</django-objects>
```

Двигатель Django Cassandra

- Установить pip: `$ pip install django-cassandra-engine`
- Добавьте «Приступая к работе» в INSTALLED_APPS в файле settings.py:
  `INSTALLED_APPS = ['django_cassandra_engine']`
- Установка Cange DATABASES Стандарт:

  Standart

```
DATABASES = {
    'default': {
        'ENGINE': 'django_cassandra_engine',
        'NAME': 'db',
        'TEST_NAME': 'test_db',
        'HOST': 'db1.example.com,db2.example.com',
        'OPTIONS': {
            'replication': {
                'strategy_class': 'SimpleStrategy',
                'replication_factor': 1
            }
        }
    }
}
```

  Cassandra создает нового пользователя cqlsh:

```
DATABASES = {
'default': {
    'ENGINE': 'django_cassandra_engine',
    'NAME': 'db',
    'TEST_NAME': 'test_db',
    'USER_NAME'='cassandradb',
    'PASSWORD'= '123cassandra',
    'HOST': 'db1.example.com,db2.example.com',
    'OPTIONS': {
        'replication': {
            'strategy_class': 'SimpleStrategy',
            'replication_factor': 1
        }
    }
}
```

}

Прочитайте Настройка базы данных онлайн: https://riptutorial.com/ru/django/topic/4933/настройка-базы-данных

глава 33: настройки

Examples

Установка часового пояса

Вы можете установить часовой пояс, который будет использоваться Django в файле `settings.py` . Примеры:

```
TIME_ZONE = 'UTC'  # use this, whenever possible
TIME_ZONE = 'Europe/Berlin'
TIME_ZONE = 'Etc/GMT+1'
```

Вот список действительных часовых поясов

*При работе в среде **Windows** это должно быть установлено так же, как и в **системном часовом поясе** .*

Если вы не хотите, чтобы Django использовал временные данные, относящиеся к часовому поясу:

```
USE_TZ = False
```

Лучшие практики Django требуют использовать `UTC` для хранения информации в базе данных:

> Даже если ваш сайт доступен только в одном часовом поясе, по-прежнему рекомендуется хранить данные в формате UTC в вашей базе данных. Основная причина - переход на летнее время (DST). Во многих странах существует система DST, где осенью осенью весной и осенью часы продвигаются вперед. Если вы работаете в местное время, вы, вероятно, столкнетесь с ошибками два раза в год, когда происходят переходы.

https://docs.djangoproject.com/en/stable/topics/i18n/timezones/

Доступ к настройкам

После того, как вы получите все свои настройки, вы захотите использовать их в своем коде. Для этого добавьте следующий импорт в свой файл:

```
from django.conf import settings
```

Затем вы можете получить доступ к своим настройкам в качестве атрибутов модуля `settings` , например:

```
if not settings.DEBUG:
    email_user(user, message)
```

Использование BASE_DIR для обеспечения мобильности приложений

Это плохая идея для жестких путей кода в вашем приложении. Всегда нужно использовать относительные URL-адреса, чтобы ваш код мог легко работать на разных машинах. Лучший способ установить это - определить такую переменную, как это

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

Затем используйте эту переменную `BASE_DIR` чтобы определить все остальные настройки.

```
TEMPLATE_PATH = os.path.join(BASE_DIR, "templates")
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),

]
```

И так далее. Это гарантирует, что вы можете переносить свой код на разные машины без каких-либо проблем.

Однако `os.path` является немного подробным. Например, если ваш модуль настроек - `project.settings.dev` , вам нужно будет написать:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(__file__)))
```

Альтернативой является использование `unipath` модуля (который вы можете установить с помощью `pip install unipath`).

```
from unipath import Path

BASE_DIR = Path(__file__).ancestor(2)  # or ancestor(3) if using a submodule

TEMPLATE_PATH = BASE_DIR.child('templates')
STATICFILES_DIRS = [
    BASE_DIR.child('static'),
]
```

Использование переменных среды для управления настройками на серверах

Использование переменных среды является широко используемым способом настройки конфигурации приложения в зависимости от его среды, как указано в приложении Twelve-Factor .

Поскольку конфигурации, вероятно, будут меняться между средами развертывания, это

очень интересный способ изменить конфигурацию без необходимости копать исходный код приложения, а также хранить секреты вне файлов приложений и репозитория исходного кода.

В Django основные настройки находятся в `settings.py` в папке вашего проекта. Поскольку это простой файл Python, вы можете использовать `os` модуль Python из стандартной библиотеки для доступа к среде (и даже иметь соответствующие значения по умолчанию).

settings.py

```
import os

SECRET_KEY = os.environ.get('APP_SECRET_KEY', 'unsafe-secret-key')

DEBUG = bool(os.environ.get('DJANGO_DEBUG', True) == 'False')

ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split()

DATABASES = {
    'default': {
        'ENGINE': os.environ.get('APP_DB_ENGINE', 'django.db.backends.sqlite3'),
        'NAME': os.environ.get('DB_NAME', 'db.sqlite'),
        'USER': os.environ.get('DB_USER', ''),
        'PASSWORD': os.environ.get('DB_PASSWORD', ''),
        'HOST': os.environ.get('DB_HOST', None),
        'PORT': os.environ.get('DB_PORT', None),
        'CONN_MAX_AGE': 600,
    }
}
```

С помощью Django вы можете изменить свою технологию баз данных, чтобы вы могли использовать sqlite3 на своей машине разработки (и это должно быть нормальным по умолчанию для передачи в систему управления версиями). Хотя это возможно, это нецелесообразно:

> Резервные службы, такие как база данных приложения, система очередей или кеш, являются одной областью, где важна четность dev / prod. (Двенадцатифакторное приложение - паритет Dev / prod)

Для использования параметра DATABASE_URL для подключения к базе данных см. Соответствующий пример .

Использование нескольких настроек

По умолчанию проект Django по умолчанию создает один `settings.py` . Часто бывает полезно разбить его следующим образом:

```
myprojectroot/
    myproject/
        __init__.py
```

```
        settings/
            __init__.py
            base.py
            dev.py
            prod.py
            tests.py
```

Это позволяет работать с разными настройками в зависимости от того, работаете ли вы в разработке, производстве, тестах или что-то еще.

При переходе от макета по умолчанию к этому макету исходные `settings.py` становятся `settings/base.py` Когда каждый другой подмодуль будет «подклассировать» `settings/base.py` , начиная с `from .base import *` . Например, вот какие `settings/dev.py` могут выглядеть так:

```
# -*- coding: utf-8 -*-
from .base import *  # noqa

DEBUG = True
INSTALLED_APPS.extend([
    'debug_toolbar',
])
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
INTERNAL_IPS = ['192.168.0.51', '192.168.0.69']
```

Альтернатива №1

Для правильной работы команд `django-admin` вам нужно будет установить переменную среды `DJANGO_SETTINGS_MODULE` (которая по умолчанию соответствует `myproject.settings`). В процессе разработки вы установите его на `myproject.settings.dev` . В процессе производства вы установите его на `myproject.settings.prod` . Если вы используете virtualenv, лучше всего установить его в сценарий `postactivate` :

```
#!/bin/sh
export PYTHONPATH="/home/me/django_projects/myproject:$VIRTUAL_ENV/lib/python3.4"
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Если вы хотите использовать модуль настроек, который не указывается `DJANGO_SETTINGS_MODULE` за один раз, вы можете использовать опцию `--settings django-admin` :

```
django-admin test --settings=myproject.settings.tests
```

Альтернатива №2

Если вы хотите оставить `DJANGO_SETTINGS_MODULE` в своей конфигурации по умолчанию (`myproject.settings`), вы можете просто сказать модулю `settings` какую конфигурацию загружать, поместив импорт в ваш файл `__init__.py` .

В приведенном выше примере тот же результат может быть достигнут с помощью набора `__init__.py` для:

```
from .dev import *
```

Использование нескольких файлов требований

Каждый файл требований должен соответствовать имени файла настроек. Прочтите несколько настроек для получения дополнительной информации.

Состав

```
djangoproject
├── config
│   ├── __init__.py
│   ├── requirements
│   │   ├── base.txt
│   │   ├── dev.txt
│   │   ├── test.txt
│   │   └── prod.txt
│   └── settings
└── manage.py
```

В файле `base.txt` установите зависимости, используемые во всех средах.

```
# base.txt
Django==1.8.0
psycopg2==2.6.1
jinja2==2.8
```

И во всех других файлах `-r base.txt` базовые зависимости с `-r base.txt` и добавьте определенные зависимости, необходимые для текущей среды.

```
# dev.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in dev env
django-queryinspect==0.1.0
```

```
# test.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in test env
nose==1.3.7
django-nose==1.4
```

```
# prod.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in production env
```

```
django-queryinspect==0.1.0
gunicorn==19.3.0
django-storages-redux==1.3
boto==2.38.0
```

Наконец, для установки зависимостей. Пример, на dev env: `pip install -r config/requirements/dev.txt`

Скрытие секретных данных с использованием файла JSON

При использовании VCS, такого как Git или SVN, есть некоторые секретные данные, которые никогда не должны быть версиями (будь то публичный или закрытый репозиторий).

Среди этих данных вы найдете параметр `SECRET_KEY` и пароль базы данных.

Общей практикой скрыть эти настройки от контроля версий является создание файла `secrets.json` в корне вашего проекта (спасибо « *Two Scoops of Django* » за идею):

```
{
    "SECRET_KEY": "N4HE:AMk:.Ader5354DR453TH8SHTQr",
    "DB_PASSWORD": "v3ry53cr3t"
}
```

И добавьте его в свой список игнорирования (`.gitignore` для git):

```
*.py[co]
*.sw[po]
*~
/secrets.json
```

Затем добавьте в свой модуль `settings` следующую функцию:

```
import json
import os
from django.core.exceptions import ImproperlyConfigured

with open(os.path.join(BASE_DIR, 'secrets.json')) as secrets_file:
    secrets = json.load(secrets_file)

def get_secret(setting, secrets=secrets):
    """Get secret setting or fail with ImproperlyConfigured"""
    try:
        return secrets[setting]
    except KeyError:
        raise ImproperlyConfigured("Set the {} setting".format(setting))
```

Затем заполните настройки таким образом:

```
SECRET_KEY = get_secret('SECRET_KEY')
DATABASES = {
    'default': {
```

```
        'ENGINE': 'django.db.backends.postgres',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': get_secret('DB_PASSWORD'),
    },
}
```

Кредиты: два совпадения Django: лучшие практики для Django 1.8, Дэниел Рой Гринфельд и Одри РойГринфельд. Copyright 2015 Two Scoops Press (ISBN 978-0981467344)

Использование DATABASE_URL из среды

На сайтах PaaS, таких как Heroku, обычно необходимо получать информацию о базе данных как единственную переменную среды URL, а не несколько параметров (хост, порт, пользователь, пароль ...).

Существует модуль `dj_database_url` который автоматически извлекает переменную среды DATABASE_URL в словарь Python, подходящий для ввода параметров базы данных в Django.

Использование:

```
import dj_database_url

if os.environ.get('DATABASE_URL'):
    DATABASES['default'] =
        dj_database_url.config(default=os.environ['DATABASE_URL'])
```

Прочитайте настройки онлайн: https://riptutorial.com/ru/django/topic/942/настройки

глава 34: Непрерывная интеграция с Дженкинсом

Examples

Jenkins 2.0+ Pipeline Script

Современные версии Jenkins (версия 2.x) поставляются с «Build Pipeline Plugin», который может использоваться для организации сложных задач CI без создания множества взаимосвязанных заданий и позволяет вам легко контролировать конфигурацию сборки / тестирования.

Вы можете установить это вручную в задании типа «Тип трубопровода», или, если ваш проект размещен в Github, вы можете использовать «GitHub Organization Folder Plugin» для автоматической настройки заданий для вас.

Вот простая конфигурация для сайтов Django, для которых требуются только указанные модули python для сайта.

```
#!/usr/bin/groovy

node {
  // If you are having issues with your project not getting updated,
  // try uncommenting the following lines.
  //stage 'Checkout'
  //checkout scm
  //sh 'git submodule update --init --recursive'

  stage 'Update Python Modules'
  // Create a virtualenv in this folder, and install or upgrade packages
  // specified in requirements.txt; https://pip.readthedocs.io/en/1.1/requirements.html
  sh 'virtualenv env && source env/bin/activate && pip install --upgrade -r requirements.txt'

  stage 'Test'
  // Invoke Django's tests
  sh 'source env/bin/activate && python ./manage.py runtests'
}
```

Jenkins 2.0+ Pipeline Script, Docker Containers

Ниже приведен пример сценария конвейера, который создает контейнер Docker, а затем запускает тесты внутри него. Предполагается, что точка входа будет либо `manage.py` либо `invoke` / `fabric` с `runtests` командой `runtests` .

```
#!/usr/bin/groovy

node {
```

```
  stage 'Checkout'
  checkout scm
  sh 'git submodule update --init --recursive'

  imageName = 'mycontainer:build'
  remotes = [
    'dockerhub-account',
  ]

  stage 'Build'
  def djangoImage = docker.build imageName

  stage 'Run Tests'
  djangoImage.run('', 'runtests')

  stage 'Push'
  for (int i = 0; i < remotes.size(); i++) {
      sh "docker tag ${imageName} ${remotes[i]}/${imageName}"
      sh "docker push ${remotes[i]}/${imageName}"
   }
}
```

Прочитайте Непрерывная интеграция с Дженкинсом онлайн:
https://riptutorial.com/ru/django/topic/5873/непрерывная-интеграция-с-дженкинсом

глава 35: Общий вид

Вступление

Общие представления представляют собой представления, которые выполняют определенное предопределенное действие, например создание, редактирование или удаление объектов, или просто отображение шаблона.

Общие представления следует отличать от функциональных представлений, которые всегда написаны вручную для выполнения требуемых задач. В двух словах можно сказать, что общие представления должны быть настроены, а функциональные представления должны быть запрограммированы.

Общие представления могут сэкономить много времени, особенно когда у вас есть много стандартизованных задач для выполнения.

замечания

Эти примеры показывают, что общие представления обычно упрощают стандартизованные задачи. Вместо того, чтобы программировать все с нуля, вы настраиваете то, что другие люди уже запрограммировали для вас. Это имеет смысл во многих ситуациях, так как позволяет больше сосредоточиться на дизайне ваших проектов, а не на процессах в фоновом режиме.

Так, если вы *всегда* будете использовать их? Нет. Они имеют смысл только в том случае, если ваши задачи достаточно стандартизированы (загрузка, редактирование, удаление объектов) и более повторяющиеся ваши задачи. Использование одного определенного общего представления только один раз, а затем переопределить все его методы для выполнения очень специфических задач, может не иметь смысла. Вам может быть лучше с функциональным представлением здесь.

Однако, если у вас много просмотров, требующих этой функции, или если ваши задачи соответствуют заданным задачам определенного общего представления, то общие представления - это именно то, что вам нужно, чтобы сделать вашу жизнь проще.

Examples

Минимальный пример: функциональные и общие представления

Пример для функционального представления для создания объекта. Исключая комментарии и пустые строки, нам нужно 15 строк кода:

```
# imports
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect

from .models import SampleObject
from .forms import SampleObjectForm

# view functioon
def create_object(request):

    # when request method is 'GET', show the template
    if request.method == GET:
        # perform actions, such as loading a model form
        form = SampleObjectForm()
        return render_to_response('template.html', locals())

    # if request method is 'POST', create the object and redirect
    if request.method == POST:
        form = SampleObjectForm(request.POST)

        # save object and redirect to success page if form is valid
        if form.is_valid:
            form.save()
            return HttpResponseRedirect('url_to_redirect_to')

        # load template with form and show errors
        else:
            return render_to_response('template.html', locals())
```

Пример для «Общего представления на основе классов» для выполнения одной и той же задачи. Нам нужно всего 7 строк кода для достижения той же задачи:

```
from django.views.generic import CreateView

from .models import SampleObject
from .forms import SampleObjectForm

class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'
```

Настройка общих представлений

Вышеприведенный пример работает только в том случае, если ваши задачи являются полностью стандартными задачами. Например, вы не добавляете дополнительный контекст.

Давайте сделаем более реалистичный пример. Предположим, мы хотим добавить заголовок страницы в шаблон. В функциональном представлении это будет работать следующим образом: только с одной дополнительной строкой:

```
def create_object(request):
    page_title = 'My Page Title'
```

```
    # ...

    return render_to_response('template.html', locals())
```

Это сложнее (или: counter-intutitive) для достижения общих представлений. Поскольку они основаны на классах, вам необходимо переопределить один или несколько методов класса для достижения желаемого результата. В нашем примере нам нужно переопределить метод *get_context_data* класса следующим образом:

```
class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context
```

Здесь нам нужно четыре дополнительных строки для кода вместо одного - по крайней мере для *первой* дополнительной переменной контекста, которую мы хотим добавить.

Общие представления с Mixins

Истинная сила общих представлений разворачивается, когда вы объединяете их с Mixins. Mixin - это просто другой класс, определенный вами, чьи методы могут быть унаследованы вашим классом вида.

Предположим, вы хотите, чтобы каждое представление отображало дополнительную переменную 'page_title' в шаблоне. Вместо того, чтобы переопределять метод get_context_data каждый раз, когда вы определяете представление, вы создаете mixin с этим методом и позволяете вашим представлениям наследовать из этого mixin. Звучит более сложно, чем на самом деле:

```
# Your Mixin
class CustomMixin(object):

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context

# Your view function now inherits from the Mixin
class CreateObject(CustomMixin, CreateView):
    model = SampleObject
    form_class = SampleObjectForm
```

```
    success_url = 'url_to_redirect_to'

# As all other view functions which need these methods
class EditObject(CustomMixin, EditView):
    model = SampleObject
    # ...
```

Красота в том, что ваш код становится гораздо более структурированным, чем в большинстве случаев с функциональными представлениями. Вся ваша логика конкретных задач находится в одном месте и только в одном месте. Кроме того, вы будете экономить огромное количество времени, особенно когда у вас много просмотров, которые всегда выполняют одни и те же задачи, за исключением разных объектов

Прочитайте Общий вид онлайн: https://riptutorial.com/ru/django/topic/9452/общий-вид

глава 36: отладка

замечания

PDB

Pdb также может распечатывать все существующие переменные в глобальной или локальной области, путем ввода `globals()` или `locals()` в (Pdb) соответственно.

Examples

Использование Python Debugger (Pdb)

Основным инструментом отладки Django является pdb , часть стандартной библиотеки Python.

Скрипт сценария Init

Рассмотрим простой скрипт `views.py` :

```
from django.http import HttpResponse


def index(request):
    foo = 1
    bar = 0

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Команда консоли для запуска сервера:

```
python manage.py runserver
```

Очевидно, что Django бросает `ZeroDivisionError` при попытке загрузить индексную страницу, но если мы притворимся, что ошибка очень глубока в коде, это может стать действительно неприятным.

Установка точки останова

К счастью, мы можем установить *точку останова* для отслеживания этой ошибки:

```
from django.http import HttpResponse

# Pdb import
import pdb
```

```
def index(request):
    foo = 1
    bar = 0

    # This is our new breakpoint
    pdb.set_trace()

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Команда консоли для запуска сервера с pdb:

```
python -m pdb manage.py runserver
```

Теперь в точке останова на странице будет вызываться запрос (Pdb) в оболочке, который также повредит ваш браузер в ожидающем состоянии.

Отладка с оболочкой pdb

Пришло время отладить это представление, взаимодействуя со скриптом через оболочку:

```
> ../views.py(12)index()
-> bug = foo/bar
# input 'foo/bar' expression to see division results:
(Pdb) foo/bar
*** ZeroDivisionError: division by zero
# input variables names to check their values:
(Pdb) foo
1
(Pdb) bar
0
# 'bar' is a source of the problem, so if we set it's value > 0...
(Pdb) bar = 1
(Pdb) foo/bar
1.0
# exception gone, ask pdb to continue execution by typing 'c':
(Pdb) c
[03/Aug/2016 10:50:45] "GET / HTTP/1.1" 200 111
```

В последней строке мы видим, что наше представление вернуло ответ `OK` и выполнило его как следует.

Чтобы остановить цикл pdb, просто введите `q` в оболочку.

Использование панели инструментов Django Debug

Во-первых, вам нужно установить панель инструментов django-debug :

```
pip install django-debug-toolbar
```

settings.py :

Затем включите его в установленные приложения проекта, но будьте осторожны - всегда полезно использовать другой файл `settings.py` для таких приложений только для разработки, а middlewares - как панель отладки:

```
# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]

MIDDLEWARE += ['debug_toolbar.middleware.DebugToolbarMiddleware']
```

Панель инструментов Debug также использует статические файлы, поэтому соответствующее приложение также должно быть включено:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.staticfiles',
    # ...
]

STATIC_URL = '/static/'

# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]
```

В некоторых случаях также необходимо установить `INTERNAL_IPS` в `settings.py` :

```
INTERNAL_IPS = ('127.0.0.1', )
```

urls.py :

В `urls.py` , как предполагает официальная документация, следующий фрагмент должен включать маршрутизацию панели отладки:

```
if settings.DEBUG and 'debug_toolbar' in settings.INSTALLED_APPS:
    import debug_toolbar
    urlpatterns += [
        url(r'^__debug__/', include(debug_toolbar.urls)),
    ]
```

Соберите статическую панель после установки:

```
python manage.py collectstatic
```

Вот так, панель инструментов debug появится на ваших страницах проекта, предоставив разнообразную полезную информацию о времени выполнения, SQL, статических файлах, сигналах и т. Д.

HTML:

Кроме того, для `django-debug-toolbar` требуется, чтобы теги *Content-type* `text/html` , `<html>` и `<body>` отображались правильно.

Если вы уверены, что настроили все правильно, но панель инструментов отладки все еще не отображается: используйте это «ядерное» решение, чтобы попытаться понять это.

Использование "assert False"

При разработке, вставляя следующую строку в свой код:

```
assert False, value
```

приведет к тому, что django поднимет `AssertionError` со значением, предоставленным как сообщение об ошибке, когда эта строка будет выполнена.

Если это происходит в представлении или в любом коде, вызванном из представления, и установлено значение `DEBUG=True` , в браузере будет отображаться полная и подробная таблица с большой информацией об отладке.

Не забудьте удалить линию, когда вы закончите!

Рассмотрите возможность написания дополнительной документации, тестов, протоколирования и утверждений вместо использования отладчика

Отладка требует времени и усилий.

Вместо того, чтобы преследовать ошибки с помощью отладчика, подумайте о том, чтобы потратить больше времени на улучшение кода:

- **Записывайте и запускайте тесты .** У Python и Django есть отличные встроенные рамки тестирования, которые могут быть использованы для проверки вашего кода намного быстрее, чем вручную с помощью отладчика.
- **Написание надлежащей документации** для ваших функций, классов и модулей. PEP 257 и Руководство по стилю Python от Google предоставляют хорошие рекомендации для написания хороших докстерий.
- **Используйте Logging** для вывода результатов из вашей программы - во время разработки и после развертывания.

- **Добавьте `assert` ионы** в свой код в важных местах: уменьшите двусмысленность, поймите проблемы по мере их создания.

Бонус: Напишите доктрины для объединения документации и тестирования!

Прочитайте отладка онлайн: https://riptutorial.com/ru/django/topic/5072/отладка

глава 37: Отношение «многие ко многим»

Examples

С сквозной моделью

```
class Skill(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Developer(models.Model):
    name = models.CharField(max_length=50)
    skills = models.ManyToManyField(Skill, through='DeveloperSkill')

class DeveloperSkill(models.Model):
    """Developer skills with respective ability and experience."""

    class Meta:
        order_with_respect_to = 'developer'
        """Sort skills per developer so that he can choose which
        skills to display on top for instance.
        """
        unique_together = [
            ('developer', 'skill'),
        ]
        """It's recommended that a together unique index be created on
        `(developer,skill)`. This is especially useful if your database is
        being access/modified from outside django. You will find that such an
        index is created by django when an explicit through model is not
        being used.
        """


    ABILITY_CHOICES = [
        (1, "Beginner"),
        (2, "Accustomed"),
        (3, "Intermediate"),
        (4, "Strong knowledge"),
        (5, "Expert"),
    ]

    developer = models.ForeignKey(Developer, models.CASCADE)
    skill = models.ForeignKey(Skill, models.CASCADE)
    """The many-to-many relation between both models is made by the
    above two foreign keys.

    Other fields (below) store information about the relation itself.
    """

    ability = models.PositiveSmallIntegerField(choices=ABILITY_CHOICES)
    experience = models.PositiveSmallIntegerField(help_text="Years of experience.")
```

Рекомендуется создать единый уникальный индекс `(developer,skill)` . Это особенно полезно, если ваша база данных получает доступ / изменена извне django. Вы обнаружите, что такой индекс создается django, когда явная сквозная модель не используется.

Простое много для многих отношений.

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Здесь мы определяем отношения, в которых у клуба есть много `Person` и членов, и Лицо может быть членом нескольких разных `Club` .

Хотя мы определяем только две модели, django фактически создает для нас три таблицы в базе данных. Это `myapp_person` , `myapp_club` и myapp_club_members. Django автоматически создает уникальный индекс в `myapp_club_members(club_id,person_id)` .

Использование полей ManyToMany

Мы используем эту модель из первого примера:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Добавить Том и Билл в ночной клуб:

```
tom = Person.objects.create(name="Tom", description="A nice guy")
bill = Person.objects.create(name="Bill", description="Good dancer")

nightclub = Club.objects.create(name="The Saturday Night Club")
nightclub.members.add(tom, bill)
```

Кто в клубе?

```
for person in nightclub.members.all():
    print(person.name)
```

Дам тебе

```
Tom
Bill
```

Прочитайте Отношение «многие ко многим» онлайн:
https://riptutorial.com/ru/django/topic/2379/отношение--многие-ко-многим-

глава 38: Отображение строк в строки с помощью HStoreField - поле PostgreSQL

Синтаксис

- FooModel.objects.filter (field_name__key_name = 'значение для запроса')

Examples

Настройка HStoreField

Во-первых, нам нужно будет сделать некоторые настройки для работы `HStoreField` .

1. убедитесь, что `django.contrib.postgres` находится в вашем `INSTALLED_APPS
2. Добавьте `HStoreExtension` в свои миграции. Не забудьте поставить `HStoreExtension` перед любыми `CreateModel` или `AddField` .

```
from django.contrib.postgres.operations import HStoreExtension
from django.db import migrations

class FooMigration(migrations.Migration):
    # put your other migration stuff here
    operations = [
        HStoreExtension(),
        ...
    ]
```

Добавление HStoreField к вашей модели

> -> Примечание: сначала убедитесь, что вы установили `HStoreField` прежде чем продолжить этот пример. (выше)

Для инициализации `HStoreField` не требуются `HStoreField` .

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Catalog(models.model):
    name = models.CharField(max_length=200)
    titles_to_authors = HStoreField()
```

Создание экземпляра новой модели

Передайте родные словарные строки для python для строк для `create()` .

```
Catalog.objects.create(name='Library of Congress', titles_to_authors={
    'Using HStoreField with Django': 'CrazyPython and la communidad',
    'Flabbergeists and thingamajigs': 'La Artista Fooista',
    'Pro Git': 'Scott Chacon and Ben Straub',
})
```

Выполнение ключевых поисков

```
Catalog.objects.filter(titles__Pro_Git='Scott Chacon and Ben Straub')
```

Использование содержит

Передайте объект `dict` аргументу `field_name__contains` в качестве аргумента ключевого слова.

```
Catalog.objects.filter(titles__contains={
        'Pro Git': 'Scott Chacon and Ben Straub'})
```

Эквивалентен оператору SQL `@>`.

Прочитайте Отображение строк в строки с помощью HStoreField - поле PostgreSQL онлайн: https://riptutorial.com/ru/django/topic/2670/отображение-строк-в-строки-с-помощью-hstorefield--поле-postgresql

глава 39: Пользовательские менеджеры и запросы

Examples

Определение базового менеджера с использованием Querysets и метода `as_manager`

Django manger - это интерфейс, через который модель django запрашивает базу данных. Поле `objects` используемое в большинстве запросов django, фактически является менеджером по умолчанию, созданным для нас django (это создается только в том случае, если мы не определяем настраиваемых менеджеров).

Почему мы должны определить пользовательский менеджер / набор запросов?

Чтобы избежать написания общих запросов по всей нашей базе кода и вместо этого ссылаться на них, используя более легкую для запоминания абстракцию. Пример. Определите, какая версия более читаема:

- Только получить всех активных пользователей: `User.objects.filter(is_active=True)` vs `User.manager.active()`
- Получить всех активных дерматологов на нашей платформе: `User.objects.filter(is_active=True).filter(is_doctor=True).filter(specialization='Dermatology')` vs `User.manager.doctors.with_specialization('Dermatology')`

Еще одно преимущество заключается в том, что если завтра мы решаем, что все `psychologists` также являются `dermatologists` , мы можем легко изменить запрос в нашем Менеджере и сделать с ним.

Ниже приведен пример создания настраиваемого `Manager` определяется путем создания `QuerySet` и использования метода `as_manager` .

```
from django.db.models.query import QuerySet

class ProfileQuerySet(QuerySet):
    def doctors(self):
        return self.filter(user_type="Doctor", user__is_active=True)

    def with_specializations(self, specialization):
        return self.filter(specializations=specialization)

    def users(self):
        return self.filter(user_type="Customer", user__is_active=True)

ProfileManager = ProfileQuerySet.as_manager
```

Мы добавим его в нашу модель, как показано ниже:

```
class Profile(models.Model):
    ...
    manager = ProfileManager()
```

ПРИМЕЧАНИЕ . Как только мы определили `manager` на нашей модели, `objects` больше не будут определены для модели.

select_related для всех запросов

Модель с ForeignKey

Мы будем работать с этими моделями:

```
from django.db import models

class Book(models.Model):
 name= models.CharField(max_length=50)
 author = models.ForeignKey(Author)

class Author(models.Model):
 name = models.CharField(max_length=50)
```

Предположим, что мы часто (всегда) `book.author.name` доступ к `book.author.name`

Ввиду

Мы могли бы использовать следующее, каждый раз,

```
books = Book.objects.select_related('author').all()
```

Но это не СУХОЙ.

Пользовательский менеджер

```
class BookManager(models.Manager):

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.select_related('author')

class Book(models.Model):
    ...
    objects = BookManager()
```

Примечание : вызов `super` должен быть изменен для python 2.x

Теперь все, что нам нужно использовать в представлениях

```
books = Book.objects.all()
```

и никаких дополнительных запросов в шаблоне / представлении не будет.

Определение пользовательских менеджеров

Очень часто приходится иметь дело с моделями, которые имеют что-то вроде `published` области. Такие типы полей почти всегда используются при извлечении объектов, так что вы обнаружите, что пишете что-то вроде:

```
my_news = News.objects.filter(published=True)
```

слишком много раз. Вы можете использовать пользовательских менеджеров для решения этих ситуаций, чтобы затем вы могли написать что-то вроде:

```
my_news = News.objects.published()
```

который является более приятным и легким для чтения другими разработчиками.

Создайте файл `managers.py` в каталоге приложения, и определить новый `models.Manager` класс:

```
from django.db import models


class NewsManager(models.Manager):

    def published(self, **kwargs):
        # the method accepts **kwargs, so that it is possible to filter
        # published news
        # i.e: News.objects.published(insertion_date__gte=datetime.now)
        return self.filter(published=True, **kwargs)
```

используйте этот класс, переопределив свойство `objects` в классе модели:

```
from django.db import models

# import the created manager
from .managers import NewsManager

class News(models.Model):
    """ News model
    """
    insertion_date = models.DateTimeField('insertion date', auto_now_add=True)
    title = models.CharField('title', max_length=255)
    # some other fields here
    published = models.BooleanField('published')

    # assign the manager class to the objects property
    objects = NewsManager()
```

Теперь вы можете получить опубликованные новости просто так:

```
my_news = News.objects.published()
```

и вы также можете выполнять большую фильтрацию:

```
my_news = News.objects.published(title__icontains='meow')
```

Прочитайте Пользовательские менеджеры и запросы онлайн:
https://riptutorial.com/ru/django/topic/1400/пользовательские-менеджеры-и-запросы

глава 40: Представления на основе классов

замечания

При использовании CBV нам часто нужно точно знать, какие методы мы можем перезаписать для каждого родового класса. На этой странице документации django перечислены все общие классы со всеми их методами сглаживания и атрибутами класса, которые мы можем использовать.

Кроме того, сайт Classy Class Based View предоставляет ту же информацию с приятным интерактивным интерфейсом.

Examples

Классные представления

Представления на основе классов позволяют сосредоточиться на том, что делает ваши взгляды особенными.

Статическая страница может не иметь ничего особенного, кроме используемого шаблона. Используйте TemplateView ! Все, что вам нужно сделать, это установить имя шаблона. Работа выполнена. Следующий.

views.py

```
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url('^about/', views.AboutView.as_view(), name='about'),
]
```

Обратите внимание, что мы не используем напрямую `AboutView` в URL- `AboutView` . Это потому, что ожидается вызываемый, и именно это `as_view()` .

Контекстные данные

Иногда вашему шаблону требуется немного больше информации. Например, мы хотели бы, чтобы пользователь в заголовке страницы со ссылкой на свой профиль рядом с линией выхода. В этих случаях используйте метод `get_context_data` .

views.py

```
class BookView(DetailView):
    template_name = "book.html"

    def get_context_data(self, **kwargs)
        """ get_context_data let you fill the template context """
        context = super(BookView, self).get_context_data(**kwargs)
        # Get Related publishers
        context['publishers'] = self.object.publishers.filter(is_active=True)
        return context
```

Вам нужно вызвать метод get_context_data для суперкласса, и он вернет экземпляр контекста по умолчанию. Любой элемент, который вы добавите в этот словарь, будет доступен для шаблона.

book.html

```
<h3>Active publishers</h3>
<ul>
  {% for publisher in publishers %}
    <li>{{ publisher.name }}</li>
  {% endfor %}
</ul>
```

Просмотр списка и сведений

Представления шаблонов подходят для статической страницы, и вы можете использовать их для всего с `get_context_data` но это будет едва лучше, чем использование функции в виде представлений.

Введите ListView и DetailView

Приложение / models.py

```
from django.db import models

class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)
    slug = models.CharField(max_length=48)
```

Приложение / views.py

```
from django.views.generic import ListView, DetailView
from .models import Pokemon


class PokedexView(ListView):
    """ Provide a list of Pokemon objects """
    model = Pokemon
    paginate_by = 25

class PokemonView(DetailView):
    model = Pokemon
```

Это все, что вам нужно, чтобы создать представление, в котором перечислены все ваши объекты моделей и представления отдельного элемента. Список даже разбит на страницы. Вы можете указать `template_name` если хотите что-то конкретное. По умолчанию он создается из имени модели.

приложение / шаблоны / приложение / pokemon_list.html

```
<!DOCTYPE html>
<title>Pokedex</title>
<ul>{% for pokemon in pokemon_list %}
    <li><a href="{% url "app:pokemon" pokemon.pk %}">{{ pokemon.name }}</a>
        &ndash; {{ pokemon.species }}
</ul>
```

Контекст заполняется списком объекта под двумя именами, `object_list` и второй `object_list` из имени модели, здесь `pokemon_list` . Если вы разместили список страниц, вам необходимо позаботиться о следующей и предыдущей ссылке. Объект Paginator может помочь в этом, он также доступен в контекстных данных.

приложение / шаблоны / приложение / pokemon_detail.html

```
<!DOCTYPE html>
<title>Pokemon {{ pokemon.name }}</title>
<h1>{{ pokemon.name }}</h1>
<h2>{{ pokemon.species }} </h2>
```

Как и прежде, контекст заполняется с моделью объекта под именем `object` и `pokemon` , второй является производным от названия модели.

Приложение / urls.py

```
from django.conf.urls import url
from . import views

app_name = 'app'
urlpatterns = [
    url(r'^pokemon/$', views.PokedexView.as_view(), name='pokedex'),
    url(r'^pokemon/(?P<pk>\d+)/$', views.PokemonView.as_view(), name='pokemon'),
]
```

В этом фрагменте URL-адрес подробного представления создается с использованием первичного ключа. Также возможно использовать slug в качестве аргумента. Это дает более привлекательный URL-адрес, который легче запомнить. Однако для этого требуется наличие поля с именем slug в вашей модели.

```
url(r'^pokemon/(?P<slug>[A-Za-z0-9_-]+)/$', views.PokemonView.as_view(), name='pokemon'),
```

Если поле с именем `slug` отсутствует, вы можете использовать параметр `slug_field` в `DetailView` чтобы указать на другое поле.

Для разбивки на страницы используйте страницу для получения параметров или поместите страницу непосредственно в URL-адрес.

Создание формы и объекта

Написание вида для создания объекта может быть довольно скучным. Вам нужно отобразить форму, вы должны ее проверить, вам нужно сохранить элемент или вернуть форму с ошибкой. Если вы не используете одно из общих представлений редактирования .

Приложение / views.py

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Pokemon


class PokemonCreate(CreateView):
```

```
    model = Pokemon
    fields = ['name', 'species']


class PokemonUpdate(UpdateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonDelete(DeleteView):
    model = Pokemon
    success_url = reverse_lazy('pokedex')
```

`CreateView` и `UpdateView` есть два обязательных атрибута, `model` и `fields` . По умолчанию оба используют имя шаблона, основанное на имени модели, дополненном «_form». Вы можете изменить только суффикс с атрибутом template_name_suffix. Перед удалением объекта DeleteView отображается сообщение с подтверждением.

Оба `UpdateView` и `DeleteView` должны извлекать объекты. Они используют тот же метод, что и `DetailView` , извлекают переменную из url и сопоставляют поля объекта.

app / templates / app / pokemon_form.html (extract)

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

`form` содержит форму со всеми необходимыми полями. Здесь он будет отображаться с абзацем для каждого поля из-за `as_p` .

app / templates / app / pokemon_confirm_delete.html (extract)

```
<form action="" method="post">
    {% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm" />
</form>
```

Тег `csrf_token` требуется из-за защиты django от подделки запроса. Действие атрибута остается пустым, поскольку URL-адрес, отображающий форму, аналогичен тому, который обрабатывает удаление / сохранение.

Остается две проблемы с моделью, если использовать то же самое, что и с примером и подробным примером. Во-первых, создание и обновление будут жаловаться на недостающий URL перенаправления. Это можно решить, добавив `get_absolute_url` к модели `get_absolute_url` . Вторая проблема - подтверждение удаления, не отображающее значимой информации. Чтобы решить эту проблему, самым простым решением является добавление строкового представления.

Приложение / models.py

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible


@python_2_unicode_compatible
class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)

    def get_absolute_url(self):
        return reverse('app:pokemon', kwargs={'pk':self.pk})

    def __str__(self):
        return self.name
```

Декоратор класса гарантирует, что все работает плавно под python 2.

Минимальный пример

views.py :

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

urls.py :

```
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

Подробнее о документации Django »

Django Class Based Views: пример CreateView

Благодаря Generic Views класса, это очень просто и легко создать CRUD-представления из наших моделей. Часто встроенный администратор Django недостаточно или не рекомендуется, и нам нужно перевернуть наши собственные представления CRUD. CBVs могут быть очень удобны в таких случаях.

Для класса `CreateView` нужны 3 вещи: модель, поля для использования и URL-адрес успеха.

Пример:

```
from django.views.generic import CreateView
from .models import Campaign

class CampaignCreateView(CreateView):
    model = Campaign
    fields = ('title', 'description')

    success_url = "/campaigns/list"
```

После успеха создания пользователь перенаправляется на `success_url` . Мы также можем определить метод `get_success_url` и использовать `reverse` или `reverse_lazy` для получения URL- `get_success_url` успеха.

Теперь нам нужно создать шаблон для этого представления. Шаблон должен быть назван в формате `<app name>/<model name>_form.html` . Название модели должно быть в нижних шапках. Например, если мое имя приложения - это `dashboard` , то для созданного выше представления мне нужно создать шаблон с именем `dashboard/campaign_form.html` .

В шаблоне переменная `form` будет содержать форму. Вот пример кода для шаблона:

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

Теперь пришло время добавить представление к нашим шаблонам url.

```
url('^campaign/new/$', CampaignCreateView.as_view(), name='campaign_new'),
```

Если мы посетим URL-адрес, мы увидим форму с полями, которые мы выбрали. Когда мы отправляем, он попытается создать новый экземпляр модели с данными и сохранить ее. При успехе пользователь будет перенаправлен на URL-адрес успеха. При ошибках форма снова будет отображаться с сообщениями об ошибках.

Один вид, несколько форм

Ниже приведен краткий пример использования нескольких форм в одном представлении Django.

```
from django.contrib import messages
from django.views.generic import TemplateView

from .forms import AddPostForm, AddCommentForm
from .models import Comment

class AddCommentView(TemplateView):

    post_form_class = AddPostForm
    comment_form_class = AddCommentForm
    template_name = 'blog/post.html'

    def post(self, request):
        post_data = request.POST or None
        post_form = self.post_form_class(post_data, prefix='post')
        comment_form = self.comment_form_class(post_data, prefix='comment')

        context = self.get_context_data(post_form=post_form,
                                        comment_form=comment_form)

        if post_form.is_valid():
            self.form_save(post_form)
        if comment_form.is_valid():
            self.form_save(comment_form)

        return self.render_to_response(context)

    def form_save(self, form):
        obj = form.save()
        messages.success(self.request, "{} saved successfully".format(obj))
        return obj

    def get(self, request, *args, **kwargs):
        return self.post(request, *args, **kwargs)
```

Прочитайте Представления на основе классов онлайн:
https://riptutorial.com/ru/django/topic/1220/представления-на-основе-классов

глава 41: Промежуточное

Вступление

Middleware в Django - это структура, которая позволяет коду подключаться к обработке ответа / запроса и изменять ввод или вывод Django.

замечания

`MIDDLEWARE_CLASSES` промежуточного уровня необходимо добавить в список settings.py `MIDDLEWARE_CLASSES` прежде чем он будет включен в выполнение. Список по умолчанию, который предоставляет Django при создании нового проекта, выглядит следующим образом:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Это все функции, которые будут выполняться в **порядке** по каждому запросу (один раз до того, как он достигнет вашего кода вида в `views.py` и один раз в обратном порядке для обратного вызова `process_response` , до версии 1.10). Они выполняют самые разнообразные функции, такие как инъекция маркера кросс-сайта Forgery (csrf) .

Порядок имеет значение, потому что, если какое-либо промежуточное программное обеспечение выполняет перенаправление, тогда все последующее промежуточное ПО никогда не запустится. Или, если промежуточное ПО ожидает, что токен csrf будет там, он должен запускаться после `CsrfViewMiddleware` .

Examples

Добавление данных в запросы

Django упрощает добавление дополнительных данных в запросы для использования в представлении. Например, мы можем разобрать субдомен на META запроса и приложить его как отдельное свойство в запросе с помощью промежуточного программного обеспечения.

```
class SubdomainMiddleware:
    def process_request(self, request):
        """
        Parse out the subdomain from the request
        """
        host = request.META.get('HTTP_HOST', '')
        host_s = host.replace('www.', '').split('.')
        request.subdomain = None
        if len(host_s) > 2:
            request.subdomain = host_s[0]
```

Если вы добавите данные с промежуточным программным обеспечением в свой запрос, вы сможете получить доступ к этим вновь добавленным данным дальше по строке. Здесь мы будем использовать проанализированный поддомен, чтобы определить что-то вроде того, какая организация обращается к вашему приложению. Этот подход полезен для приложений, которые развернуты с настройкой DNS с поддоменами подстановочных знаков, которые указывают на один экземпляр, и человек, обращающийся к приложению, хочет, чтобы сдержанная версия зависела от точки доступа.

```
class OrganizationMiddleware:
    def process_request(self, request):
        """
        Determine the organization based on the subdomain
        """
        try:
            request.org = Organization.objects.get(domain=request.subdomain)
        except Organization.DoesNotExist:
            request.org = None
```

Помните, что порядок имеет значение, когда промежуточное программное обеспечение зависит друг от друга. Для запросов вы хотите, чтобы зависимое промежуточное ПО было размещено после зависимости.

```
MIDDLEWARE_CLASSES = [
    ...
    'myapp.middleware.SubdomainMiddleware',
    'myapp.middleware.OrganizationMiddleware',
    ...
]
```

Среднее ПО для фильтрации по IP-адресу

Сначала: структура пути

Если у вас его нет, вам нужно создать папку **промежуточного программного обеспечения** в своем приложении, следуя структуре:

```
yourproject/yourapp/middleware
```

Средство промежуточного содержимого папки должно быть помещено в ту же папку, что и

settings.py, urls, templates ...

Важно: Не забудьте создать пустой файл инициализации .py внутри папки промежуточного программного обеспечения, чтобы ваше приложение распознало эту папку

Вместо того, чтобы иметь отдельную папку, содержащую ваши классы промежуточного программного обеспечения, также можно разместить свои функции в одном файле - `yourproject/yourapp/middleware.py` .

Второе: создание промежуточного программного обеспечения

Теперь мы должны создать файл для нашего специального промежуточного программного обеспечения. В этом примере предположим, что мы хотим, чтобы промежуточное программное обеспечение, которое фильтрует пользователей на основе их IP-адреса, мы создаем файл **filter_ip_middleware.py** :

```
#yourproject/yourapp/middleware/filter_ip_middleware.py
from django.core.exceptions import PermissionDenied

class FilterIPMiddleware(object):
    # Check if client IP address is allowed
    def process_request(self, request):
        allowed_ips = ['192.168.1.1', '123.123.123.123', etc...] # Authorized ip's
        ip = request.META.get('REMOTE_ADDR') # Get client IP address
        if ip not in allowed_ips:
            raise PermissionDenied # If user is not allowed raise Error

       # If IP address is allowed we don't do anything
       return None
```

Третье: добавьте промежуточное ПО в наш 'settings.py'

Нам нужно искать `MIDDLEWARE_CLASSES` внутри settings.py, и там нам нужно добавить наше промежуточное ПО (*добавьте его в последнюю позицию*). Это должно выглядеть так:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
     # Above are Django standard middlewares

     # Now we add here our custom middleware
     'yourapp.middleware.filter_ip_middleware.FilterIPMiddleware'
)
```

Готово! Теперь каждый запрос от каждого клиента вызовет ваше собственное промежуточное программное обеспечение и обработает ваш собственный код!

Исключительное исключение

Скажем, вы внедрили некоторую логику для обнаружения попыток изменения объекта в базе данных, в то время как клиент, который представил изменения, не вносил последних изменений. Если такой случай случается, вы `ConfictError(detailed_message)` пользовательское исключение `ConfictError(detailed_message)` .

Теперь вы хотите вернуть код статуса HTTP 409 (Confict) при возникновении этой ошибки. Обычно вы можете использовать в качестве промежуточного программного обеспечения для этого, вместо того чтобы обрабатывать его в каждом представлении, которое может вызвать это исключение.

```
class ConfictErrorHandlingMiddleware:
    def process_exception(self, request, exception):
        if not isinstance(exception, ConflictError):
            return  # Propagate other exceptions, we only handle ConflictError
        context = dict(confict_details=str(exception))
        return TemplateResponse(request, '409.html', context, status=409)
```

Понимание нового стиля промежуточного ПО Django 1.10

Django 1.10 представил новый стиль промежуточного `process_request` котором `process_request` и `process_response` объединяются вместе.

В этом новом стиле *промежуточное программное обеспечение является вызываемым, которое возвращает другой вызываемый* . Ну, на самом деле **первый является промежуточным заводом,** а **последний является промежуточным программным обеспечением** .

Завод *промежуточного ПО* принимает в качестве единственного аргумента следующее *промежуточное программное обеспечение* в стеке middlewares или само представление, когда достигается дно стека.

Среднее ПО принимает запрос как единственный аргумент и **всегда возвращает `HttpResponse` .**

Лучший пример для иллюстрации того, как работает *промежуточное программное обеспечение* нового стиля, вероятно, показывает, как создать *промежуточное программное обеспечение,* совместимое с обратной *связью* :

```
class MyMiddleware:

    def __init__(self, next_layer=None):
        """We allow next_layer to be None because old-style middlewares
        won't accept any argument.
        """
        self.get_response = next_layer
```

```
    def process_request(self, request):
        """Let's handle old-style request processing here, as usual."""
        # Do something with request
        # Probably return None
        # Or return an HttpResponse in some cases

    def process_response(self, request, response):
        """Let's handle old-style response processing here, as usual."""
        # Do something with response, possibly using request.
        return response

    def __call__(self, request):
        """Handle new-style middleware here."""
        response = self.process_request(request)
        if response is None:
            # If process_request returned None, we must call the next middleware or
            # the view. Note that here, we are sure that self.get_response is not
            # None because this method is executed only in new-style middlewares.
            response = self.get_response(request)
        response = self.process_response(request, response)
        return response
```

Прочитайте Промежуточное онлайн: https://riptutorial.com/ru/django/topic/1721/промежуточное

глава 42: Просмотры

Вступление

Функция просмотра или короткое представление - это просто функция Python, которая принимает веб-запрос и возвращает ответ Web. -Django Documentation-

Examples

[Вводный] Простой вид (Hello World Equivalent)

Давайте создадим очень простое представление, чтобы ответить шаблону «Hello World» в формате html.

1. Для этого перейдите в `my_project/my_app/views.py` (здесь мы разместим наши функции просмотра) и добавим следующий вид:

```
from django.http import HttpResponse

def hello_world(request):
    html = "<html><title>Hello World!</title><body>Hello World!</body></html>"
    return HttpResponse(html)
```

2. Чтобы вызвать это представление, нам нужно настроить шаблон url в `my_project/my_app/urls.py` :

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^hello_world/$', views.hello_world, name='hello_world'),
]
```

3. Запустите сервер: `python manage.py runserver`

   Теперь, если мы `http://localhost:8000/hello_world/` , наш шаблон (строка html) будет отображаться в нашем браузере.

Прочитайте Просмотры онлайн: https://riptutorial.com/ru/django/topic/7490/просмотры

глава 43: развертывание

Examples

Запуск приложения Django с помощью Gunicorn

1. Установить пушки

   ```
   pip install gunicorn
   ```

2. Из папки проекта django (в той же папке, где находится manage.py), выполните следующую команду для запуска текущего проекта django с помощью gunicorn

   ```
   gunicorn [projectname].wsgi:application -b 127.0.0.1:[port number]
   ```

   Вы можете использовать опцию `--env` чтобы установить путь для загрузки настроек

   ```
   gunicorn --env DJANGO_SETTINGS_MODULE=[projectname].settings [projectname].wsgi
   ```

   или запускается как процесс демона с использованием опции `-D`

3. После успешного запуска пушки, в консоли появятся следующие строки

   ```
   Starting gunicorn 19.5.0

   Listening at: http://127.0.0.1:[port number] ([pid])
   ```

   .... (другая дополнительная информация о сервере пушки)

Развертывание с Heroku

1. Загрузить Heroku Toolbelt .

2. Перейдите в корень источников вашего приложения Django. Вам понадобится tk

3. Тип `heroku create [app_name]` . Если вы не укажете имя приложения, Heroku будет случайным образом генерировать его для вас. URL вашего приложения будет `http://[app name].herokuapp.com`

4. Создайте текстовый файл с именем `Procfile` . Не помещайте расширение в конец.

   ```
   web: <bash command to start production server>
   ```

   > Если у вас есть рабочий процесс, вы можете добавить его тоже. Добавьте еще одну строку в формате: `worker-name: <bash command to start worker>`

5. Добавьте файл requirements.txt.

- Если вы используете виртуальную среду, запустите `pip freeze > requirements.txt`
- В противном случае, *получите виртуальную среду!* , Вы также можете вручную указать пакеты Python, которые вам нужны, но это не будет описано в этом уроке.

6. **Это время развертывания!**

    1. `git push heroku master`

       Heroku нуждается в репозитории git или папке Dropbox для развертывания. Вы также можете настроить автоматическую перезагрузку из репозитория GitHub на сайте `heroku.com` , но мы не будем `heroku.com` этом в этом уроке.

    2. `heroku ps:scale web=1`

       Это увеличивает количество «динамиков» в сети. Здесь вы можете узнать больше о динамиках .

    3. `heroku open` или перейти на `http://app-name.herokuapp.com`

       Совет: `heroku open` открывает URL-адрес вашего приложения heroku в браузере по умолчанию.

7. Добавьте **дополнения** . Вам нужно настроить приложение Django для привязки к базам данных, предоставленным в Heroku, в качестве «надстроек». Этот пример не охватывает этого, но еще один пример находится в стадии разработки при развертывании баз данных в Heroku.

Простое удаленное развертывание fabfile.py

Fabric - это библиотека Python (2.5-2.7) и средство командной строки для оптимизации использования SSH для задач развертывания приложений или системного администрирования. Он позволяет выполнять произвольные функции Python через командную строку.

Установить ткань через `pip install fabric`
Создайте `fabfile.py` в корневом каталоге:

```
#myproject/fabfile.py
from fabric.api import *

@task
def dev():
    # details of development server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def release():
```

```
    # details of release server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def run():
    with cd('path/to/your_project/'):
        with prefix('source ../env/bin/activate'):
        # activate venv, suppose it appear in one level higher
            # pass commands one by one
            run('git pull')
            run('pip install -r requirements.txt')
            run('python manage.py migrate --noinput')
            run('python manage.py collectstatic --noinput')
            run('touch reload.txt')
```

Чтобы выполнить файл, просто используйте команду `fab` :

```
$ fab dev run  # for release server, `fab release run`
```

Примечание. Вы не можете настроить ssh-ключи для github и просто ввести логин и пароль вручную, в то время как fabfile работает, то же самое с ключами.

Использование шаблона Starter Heroku Django.

Если вы планируете разместить свой сайт Django на Heroku, вы можете начать свой проект с использованием шаблона Starter Heroku Django:

```
django-admin.py startproject --template=https://github.com/heroku/heroku-django-
template/archive/master.zip --name=Procfile YourProjectName
```

Он имеет готовые готовые конфигурации для статических файлов, настроек базы данных, Gunicorn и т. Д. И улучшений для использования функций статического файла Django через WhiteNoise. Это сэкономит ваше время, это All-Ready для хостинга на Heroku, Просто создайте свой сайт в верхней части этого шаблона

Чтобы развернуть этот шаблон на Heroku:

```
git init
git add -A
git commit -m "Initial commit"

heroku create
git push heroku master

heroku run python manage.py migrate
```

Это оно!

Инструкции по развертыванию Django. Nginx + Gunicorn + Supervisor для Linux (Ubuntu)

Три основных инструмента.

1. nginx - бесплатный, с открытым исходным кодом, высокопроизводительный HTTP-сервер и обратный прокси, с высокой производительностью;
2. gunicorn - «Зеленый единорог» - это HTTP-сервер Python WSGI для UNIX (необходимый для управления вашим сервером);
3. supervisor - система клиент / сервер, которая позволяет своим пользователям контролировать и контролировать ряд процессов в UNIX-подобных операционных системах. Используется при сбое приложения или системы, перезапускает вашу камеру django / celery / celery, и т. Д .;

Чтобы сделать это простым, предположим, что ваше приложение находится в этом каталоге: `/home/root/app/src/` и мы будем использовать пользователя `root` (но вы должны создать отдельного пользователя для своего приложения). Также наша виртуальная среда будет расположена в `/home/root/app/env/` path.

NGINX

Начнем с nginx. Если nginx еще не установлен на машине, установите его с помощью `sudo apt-get install nginx` . Позже вы должны создать новый файл конфигурации в вашем каталоге nginx `/etc/nginx/sites-enabled/yourapp.conf` . Если есть файл с именем `default.conf` - удалите его.

Вставьте код в файл конфига nginx, который попытается запустить вашу службу с помощью файла сокета; Позже будет конфигурация пушки. Файл сокета используется здесь для связи между nginx и gunicorn. Это также можно сделать с помощью портов.

```
# your application name; can be whatever you want
upstream yourappname {
    server        unix:/home/root/app/src/gunicorn.sock fail_timeout=0;
}

server {
    # root folder of your application
    root        /home/root/app/src/;

    listen        80;
    # server name, your main domain, all subdomains and specific subdomains
    server_name   yourdomain.com *.yourdomain.com somesubdomain.yourdomain.com

    charset       utf-8;

    client_max_body_size                        100m;

    # place where logs will be stored;
    # folder and files have to be already located there, nginx will not create
```

```
    access_log        /home/root/app/src/logs/nginx-access.log;
    error_log         /home/root/app/src/logs/nginx-error.log;

    # this is where your app is served (gunicorn upstream above)
    location / {
        uwsgi_pass  yourappname;
        include     uwsgi_params;
    }

    # static files folder, I assume they will be used
    location /static/ {
        alias         /home/root/app/src/static/;
    }

    # media files folder
    location /media/ {
        alias         /home/root/app/src/media/;
    }

}
```

GUNICORN

Теперь наш скрипт GUNICORN, который будет отвечать за запуск приложения django на сервере. Прежде всего, нужно установить пушки в виртуальную среду с помощью `pip install gunicorn` .

```
#!/bin/bash

ME="root"
DJANGODIR=/home/root/app/src # django app dir
SOCKFILE=/home/root/app/src/gunicorn.sock # your sock file - do not create it manually
USER=root
GROUP=webapps
NUM_WORKERS=3
DJANGO_SETTINGS_MODULE=yourapp.yoursettings
DJANGO_WSGI_MODULE=yourapp.wsgi
echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR

source /home/root/app/env/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Gunicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not use --
daemon)
exec /home/root/app/env/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name root \
  --workers $NUM_WORKERS \
  --user=$USER --group=$GROUP \
```

```
  --bind=unix:$SOCKFILE \
  --log-level=debug \
  --log-file=-
```

чтобы иметь возможность запускать сценарий запуска пушки, он должен иметь режим выполнения, поэтому

`sudo chmod u+x /home/root/app/src/gunicorn_start`

теперь вы сможете запустить свой сервер для пушек, просто используя `./gunicorn_start`

РУКОВОДИТЕЛЬ

Как сказано в начале, мы хотим, чтобы наше приложение было перезапущено при сбое супервизора. Если диспетчер еще не установлен на сервере с помощью `sudo apt-get install supervisor` .

Сначала установите диспетчер. Затем создайте файл `.conf` в основном каталоге `/etc/supervisor/conf.d/your_conf_file.conf`

содержимое конфигурационного файла:

```
[program:yourappname]
command = /home/root/app/src/gunicorn_start
user = root
stdout_logfile = /home/root/app/src/logs/gunicorn_supervisor.log
redirect_stderr = true
```

Краткая инструкция, `[program:youappname]` требуется в начале, это будет наш идентификатор. также `stdout_logfile` - это файл, в котором будут храниться журналы, как доступ, так и ошибки.

Сделав это, мы должны сказать нашему супервизору, что мы только что добавили новый файл конфигурации. Для этого существует другой процесс для другой версии Ubuntu.

Для `Ubuntu version 14.04 or lesser` , просто запустите эти команды:

`sudo supervisorctl reread` reread -> перечитывает все файлы конфигурации внутри каталога супервизора, это должно распечатывать: **yourappname: доступно**

`sudo supervisorctl update` -> обновляет супервизор до новых добавленных файлов конфигурации; должен распечатать **yourappname: добавлена группа процессов**

Для `Ubuntu 16.04` Запуск:

```
sudo service supervisor restart
```

и чтобы проверить, правильно ли работает приложение, выполните

```
sudo supervisorctl status yourappname
```

Это должно отображать:

```
yourappname RUNNING pid 18020, uptime 0:00:50
```

Чтобы получить живую демонстрацию этой процедуры, просмотрите это **видео** .

Развертывание локально без настройки apache / nginx

Рекомендуемый способ развертывания приложений требует использования Apache / Nginx для обслуживания статического контента. Таким образом, когда `DEBUG` ошибочно статична, а содержимое мультимедиа не загружается. Однако мы можем загружать статический контент в развертывание без необходимости установки сервера Apache / Nginx для нашего приложения, используя:

```
python manage.py runserver --insecure
```

Это предназначено только для локального развертывания (например, LAN) и никогда не должно использоваться в производстве и доступно только в том случае, если приложение `staticfiles` находится в настройке `INSTALLED_APPS` вашего проекта.

Прочитайте развертывание онлайн: https://riptutorial.com/ru/django/topic/2792/развертывание

глава 44: Расширение или замена модели пользователя

Examples

Пользовательская модель пользователя с адресом электронной почты в качестве основного поля входа.

models.py:

```
from __future__ import unicode_literals
from django.db import models
from django.contrib.auth.models import (
        AbstractBaseUser, BaseUserManager, PermissionsMixin)
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _


class UserManager(BaseUserManager):
    def _create_user(self, email,password, is_staff, is_superuser, **extra_fields):
        now = timezone.now()
        if not email:
            raise ValueError('users must have an email address')
        email = self.normalize_email(email)
        user = self.model(email = email,
                          is_staff = is_staff,
                          is_superuser = is_superuser,
                          last_login = now,
                          date_joined = now,
                          **extra_fields)
        user.set_password(password)
        user.save(using = self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        user = self._create_user(email, password, False, False, **extra_fields)
        return user

    def create_superuser(self, email, password, **extra_fields):
        user = self._create_user(email, password, True, True, **extra_fields)
        return user

class User(AbstractBaseUser,PermissionsMixin):
    """My own custom user class"""

    email = models.EmailField(max_length=255, unique=True, db_index=True,
verbose_name=_('email address'))
    date_joined = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()
```

```
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')

    def get_full_name(self):
    """Return the email."""
        return self.email

    def get_short_name(self):
    """Return the email."""
        return self.email
```

forms.py:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from .models import User


class RegistrationForm(UserCreationForm):
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email'}),
        label="Email")
    password1 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':'password1'}),
        label="Password")
    password2 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name': 'password2'}),
        label="Password (again)")

    '''added attributes so as to customise for styling, like bootstrap'''
    class Meta:
        model = User
        fields = ['email','password1','password2']
        field_order = ['email','password1','password2']

    def clean(self):
    """
    Verifies that the values entered into the password fields match
    NOTE : errors here will appear in 'non_field_errors()'
    """
        cleaned_data = super(RegistrationForm, self).clean()
        if 'password1' in self.cleaned_data and 'password2' in self.cleaned_data:
            if self.cleaned_data['password1'] != self.cleaned_data['password2']:
                raise forms.ValidationError("Passwords don't match. Please try again!")
        return self.cleaned_data

    def save(self, commit=True):
        user = super(RegistrationForm,self).save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        if commit:
            user.save()
        return user

#The save(commit=False) tells Django to save the new record, but dont commit it to the
database yet
```

```
class AuthenticationForm(forms.Form): # Note: forms.Form NOT forms.ModelForm
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email','placeholder':'Email'}),
        label='Email')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':
'password','placeholder':'Password'}),
        label='Password')

    class Meta:
        fields = ['email', 'password']
```

views.py:

```
from django.shortcuts import redirect, render, HttpResponse
from django.contrib.auth import login as django_login, logout as django_logout, authenticate
as django_authenticate
#importing as such so that it doesn't create a confusion with our methods and django's default
methods

from django.contrib.auth.decorators import login_required
from .forms import AuthenticationForm, RegistrationForm


def login(request):
    if request.method == 'POST':
        form = AuthenticationForm(data = request.POST)
        if form.is_valid():
            email = request.POST['email']
            password = request.POST['password']
            user = django_authenticate(email=email, password=password)
            if user is not None:
                if user.is_active:
                    django_login(request,user)
                    return redirect('/dashboard') #user is redirected to dashboard
    else:
        form = AuthenticationForm()

    return render(request,'login.html',{'form':form,})

def register(request):
    if request.method == 'POST':
        form = RegistrationForm(data = request.POST)
        if form.is_valid():
            user = form.save()
            u = django_authenticate(user.email = user, user.password = password)
            django_login(request,u)
            return redirect('/dashboard')
    else:
        form = RegistrationForm()

    return render(request,'register.html',{'form':form,})

def logout(request):
    django_logout(request)
    return redirect('/')

@login_required(login_url ="/")
def dashboard(request):
    return render(request, 'dashboard.html',{})
```

settings.py:

```
AUTH_USER_MODEL = 'myapp.User'
```

admin.py

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import Group
from .models import User


class UserAdmin(BaseUserAdmin):
    list_display = ('email','is_staff')
    list_filter = ('is_staff',)
    fieldsets = ((None,
                  {'fields':('email','password')}), ('Permissions',{'fields':('is_staff',)}),)
    add_fieldsets = ((None, {'classes': ('wide',), 'fields': ('email', 'password1',
'password2')}),)
    search_fields =('email',)
    ordering = ('email',)
    filter_horizontal = ()

admin.site.register(User, UserAdmin)
admin.site.unregister(Group)
```

Используйте `email` как имя пользователя и избавитесь от поля` username '

Если вы хотите избавиться от поля `username` и использовать `email` качестве уникального идентификатора пользователя, вам нужно будет создать `User` модель, расширяющую `AbstractBaseUser` вместо `AbstractUser` . Действительно, `username` и `email` определены в `AbstractUser` и вы не можете их переопределить. Это означает, что вам также придется переопределять все поля, которые вы хотите определить в `AbstractUser` .

```
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin, BaseUserManager,
)
from django.db import models
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _

class UserManager(BaseUserManager):

    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user
```

```
    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

    return self._create_user(email, password, **extra_fields)


class User(AbstractBaseUser, PermissionsMixin):
    """PermissionsMixin contains the following fields:
        - `is_superuser`
        - `groups`
        - `user_permissions`
     You can omit this mix-in if you don't want to use permissions or
     if you want to implement your own permissions logic.
     """

    class Meta:
        verbose_name = _("user")
        verbose_name_plural = _("users")
        db_table = 'auth_user'
        # `db_table` is only needed if you move from the existing default
        # User model to a custom one. This enables to keep the existing data.

    USERNAME_FIELD = 'email'
    """Use the email as unique username."""

    REQUIRED_FIELDS = ['first_name', 'last_name']

    GENDER_MALE = 'M'
    GENDER_FEMALE = 'F'
    GENDER_CHOICES = [
        (GENDER_MALE, _("Male")),
        (GENDER_FEMALE, _("Female")),
    ]

    email = models.EmailField(
        verbose_name=_("email address"), unique=True,
        error_messages={
            'unique': _(
                "A user is already registered with this email address"),
        },
    )
    gender = models.CharField(
        max_length=1, blank=True, choices=GENDER_CHOICES,
        verbose_name=_("gender"),
    )
    first_name = models.CharField(
        max_length=30, verbose_name=_("first name"),
    )
    last_name = models.CharField(
        max_length=30, verbose_name=_("last name"),
```

```
    )
    is_staff = models.BooleanField(
        verbose_name=_("staff status"),
        default=False,
        help_text=_(
            "Designates whether the user can log into this admin site."
        ),
    )
    is_active = models.BooleanField(
        verbose_name=_("active"),
        default=True,
        help_text=_(
            "Designates whether this user should be treated as active. "
            "Unselect this instead of deleting accounts."
        ),
    )
    date_joined = models.DateTimeField(
        verbose_name=_("date joined"), default=timezone.now,
    )

    objects = UserManager()
```

Расширить модель пользователя Django

Наш класс `UserProfile`

Создайте `UserProfile` модели `UserProfile` с отношением `OneToOne` к модели `User` по умолчанию:

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    photo = FileField(verbose_name=_("Profile Picture"),
                      upload_to=upload_to("main.UserProfile.photo", "profiles"),
                      format="Image", max_length=255, null=True, blank=True)
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)
    phone = models.CharField(max_length=20, blank=True, default='')
    city = models.CharField(max_length=100, default='', blank=True)
    country = models.CharField(max_length=100, default='', blank=True)
    organization = models.CharField(max_length=100, default='', blank=True)
```

Django Сигналы на работе

Используя Django Signals, создайте новый `UserProfile` сразу же, `UserProfile` объект `User` . Эта функция может быть `UserProfile` под `UserProfile` модели `UserProfile` в том же файле или размещать ее там, где вам нравится. Меня не волнует, так же, как вы его правильно ссылаетесь.

```
def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
```

```
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

`inlineformset_factory` для спасения

Теперь для вашего `views.py` вы можете иметь что-то вроде этого:

```
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from .models import UserProfile
from .forms import UserForm
from django.forms.models import inlineformset_factory
from django.core.exceptions import PermissionDenied
@login_required() # only logged in users should access this
def edit_user(request, pk):
    # querying the User object with pk from url
    user = User.objects.get(pk=pk)

    # prepopulate UserProfileForm with retrieved user values from above.
    user_form = UserForm(instance=user)

    # The sorcery begins from here, see explanation https://blog.khophi.co/extending-django-
user-model-userprofile-like-a-pro/
    ProfileInlineFormset = inlineformset_factory(User, UserProfile, fields=('website', 'bio',
'phone', 'city', 'country', 'organization'))
    formset = ProfileInlineFormset(instance=user)

    if request.user.is_authenticated() and request.user.id == user.id:
        if request.method == "POST":
            user_form = UserForm(request.POST, request.FILES, instance=user)
            formset = ProfileInlineFormset(request.POST, request.FILES, instance=user)

            if user_form.is_valid():
                created_user = user_form.save(commit=False)
                formset = ProfileInlineFormset(request.POST, request.FILES,
instance=created_user)

                if formset.is_valid():
                    created_user.save()
                    formset.save()
                    return HttpResponseRedirect('/accounts/profile/')

        return render(request, "account/account_update.html", {
            "noodle": pk,
            "noodle_form": user_form,
            "formset": formset,
        })
    else:
        raise PermissionDenied
```

Наш шаблон

Затем переместите все на свой шаблон `account_update.html` так:

```
{% load material_form %}
<!-- Material form is just a materialize thing for django forms -->
<div class="col s12 m8 offset-m2">
```

```
      <div class="card">
        <div class="card-content">
        <h2 class="flow-text">Update your information</h2>
          <form action="." method="POST" class="padding">
            {% csrf_token %} {{ noodle_form.as_p }}
            <div class="divider"></div>
            {{ formset.management_form }}
                {{ formset.as_p }}
            <button type="submit" class="btn-floating btn-large waves-light waves-effect"><i
class="large material-icons">done</i></button>
            <a href="#" onclick="window.history.back(); return false;" title="Cancel"
class="btn-floating waves-effect waves-light red"><i class="material-icons">history</i></a>

        </form>
        </div>
    </div>
</div>
```

Выделенный фрагмент из расширенного пользовательского файла Django, такого как Pro

Спецификация пользовательской модели пользователя

Встроенная `User` модель Django не всегда подходит для некоторых проектов. На некоторых сайтах может быть больше смысла использовать адрес электронной почты вместо имени пользователя.

Вы можете переопределить модель `User` по умолчанию, добавив вашу `User` модель `User` в параметр `AUTH_USER_MODEL` в файле настроек проектов:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Обратите внимание, что настоятельно `AUTH_USER_MODEL` создать `AUTH_USER_MODEL` прежде чем создавать какие-либо миграции или запускать `manage.py migrate` в первый раз. Из-за ограничений функции динамической зависимости Django.

Например, в вашем блоге вы можете захотеть, чтобы другие авторы имели возможность входа в систему с адресом электронной почты вместо обычного имени пользователя, поэтому мы создаем `User` модель `User` с адресом электронной почты как `USERNAME_FIELD` :

```
from django.contrib.auth.models import AbstractBaseUser

class CustomUser(AbstractBaseUser):
     email = models.EmailField(unique=True)

     USERNAME_FIELD = 'email'
```

Наследуя `AbstractBaseUser` мы можем построить совместимую модель `User` . `AbstractBaseUser` обеспечивает основную реализацию модели `User` .

Чтобы команда Django `manage.py createsuperuser` знала, какие другие поля необходимы, мы можем указать `REQUIRED_FIELDS` . Это значение не влияет на другие части Django!

```
class CustomUser(AbstractBaseUser):
    ...
    first_name = models.CharField(max_length=254)
    last_name = models.CharField(max_length=254)
    ...
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

Чтобы быть совместимым с другой частью Django, нам все равно нужно указать значение `is_active` , функции `get_full_name()` и `get_short_name()` :

```
class CustomUser(AbstractBaseUser):
    ...
    is_active = models.BooleanField(default=False)
    ...
    def get_full_name(self):
        full_name = "{0} {1}".format(self.first_name, self.last_name)
        return full_name.strip()

    def get_short_name(self):
        return self.first_name
```

Вы также должны создать пользовательский `UserManager` для своей модели `User` , который позволяет Django использовать функции `create_user()` и `create_superuser()` :

```
from django.contrib.auth.models import BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, first_name, last_name, password=None):
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
        )

        user.set_password(password)
        user.first_name = first_name
        user.last_name = last_name
        user.save(using=self._db)
        return user

    def create_superuser(self, email, first_name, last_name, password):
        user = self.create_user(
            email=email,
            first_name=first_name,
            last_name=last_name,
            password=password,
        )

        user.is_admin = True
        user.is_active = True
        user.save(using=self.db)
        return user
```

Ссылка на модель пользователя

Ваш код не будет работать в проектах, где вы ссылаетесь на модель `User` (*и где значение* `AUTH_USER_MODEL` *было изменено*) напрямую.

Например: если вы хотите создать `Post` модель для блога с `User` моделью `User` , вы должны указать `User` модель `User` следующим образом:

```
from django.conf import settings
from django.db import models

class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Прочитайте Расширение или замена модели пользователя онлайн:
https://riptutorial.com/ru/django/topic/1209/расширение-или-замена-модели-пользователя

глава 45: сигналы

параметры

Класс / Метод	Почему?
Класс UserProfile ()	Класс UserProfile расширяет модель пользователя Django по умолчанию .
Метод create_profile ()	Метод create_profile () выполняется, когда `post_save` пользовательской моделью `post_save` .

замечания

Теперь детали.

Сигналы Django - это способ информировать ваше приложение о некоторых задачах (например, о модели до или после сохранения или удаления), когда это происходит.

Эти сигналы позволяют вам немедленно выполнять действия по вашему выбору, когда сигнал высвобождается.

Например, в **любое время** , **когда** создается новый пользователь Django, Пользовательская модель выпускает сигнал с ассоциированными параметрами, такими как `sender=User` что позволяет вам конкретно настроить ваше прослушивание сигналов на конкретную деятельность, которая происходит, в этом случае новое создание пользователя ,

В приведенном выше примере намерение состоит в создании объекта UserProfile *сразу* после создания объекта User. Поэтому, прослушивая сигнал `post_save` из модели `User` (по умолчанию Django User Model), мы создаем объект `UserProfile` сразу после создания нового `User` .

Документация Django предоставляет обширную документацию по всем возможным сигналам .

Тем не менее, приведенный выше пример поясняет на практике типичный пример использования при использовании сигналов может быть полезным дополнением.

"С большой властью приходит большая ответственность". Может возникнуть соблазн иметь сигналы, разбросанные по всему вашему приложению или проекту, только потому, что они потрясающие. Ну, не надо. Потому что они классные, не делают их идеальным решением для каждой простой ситуации, которая приходит на ум.

Сигналы отлично подходят, как обычно, не для всех. Вход / Выход, сигналы отличные. Ключевые модели, выпускающие знаки, такие как User Model, если они прекрасны.

Создание сигналов для каждой модели в вашем приложении может быть подавляющим в какой-то момент и победить всю идею спарringового использования сигналов Django.

Не используйте сигналы, когда (на основе Two Scoops of Django book):

- Сигнал относится к одной конкретной модели и может быть перенесен в один из методов этой модели, возможно, вызванный функцией `save()` .
- Сигнал можно заменить специальным методом диспетчера модели.
- Сигнал относится к определенному виду и может быть перемещен в эту точку зрения

Можно использовать сигналы, когда:

- Ваш приемник сигналов должен внести изменения в несколько моделей.
- Вы хотите отправлять один и тот же сигнал из нескольких приложений и обрабатывать их одинаковым способом с помощью обычного приемника.
- Вы хотите аннулировать кеш после сохранения модели.
- У вас необычный сценарий, который требует обратного вызова, и нет другого способа справиться с ним, кроме использования сигнала. Например, вы хотите вызвать что-то на основе `save()` или `init()` модели стороннего приложения. Вы не можете изменить сторонний код, и его расширение может быть невозможным, поэтому сигнал обеспечивает триггер для обратного вызова.

Examples

Расширение примера профиля пользователя

Этот пример представляет собой фрагмент, взятый из расширенного профиля пользователя Django, такого как Pro

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

Разный синтаксис для отправки / предварительного сигнала

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

@receiver(post_save, sender=UserProfile)
def post_save_user(sender, **kwargs):
    user = kwargs.get('instance')
    if kwargs.get('created'):
        ...
```

Как найти, является ли это вставкой или обновлением в сигнале pre_save

Используя `pre_save` мы можем определить, было ли действие `save` в нашей базе данных об обновлении существующего объекта или создании нового.

Для этого вы можете проверить состояние объекта модели:

```
    @receiver(pre_save, sender=User)
    def pre_save_user(sender, instance, **kwargs):
        if not instance._state.adding:
            print ('this is an update')
        else:
            print ('this is an insert')
```

Теперь каждый раз, когда выполняется действие `save` , будет `pre_save` сигнал `pre_save` и будет печатать:

- `this is an update` если действие происходит от действия обновления.
- `this is an insert` если действие происходит от действия вставки.

Обратите внимание: этот метод не требует дополнительных запросов к базе данных.

Наследование сигналов на расширенных моделях

Сигналы Django ограничиваются точными сигнатурами класса при регистрации, и поэтому подклассы моделей не сразу регистрируются на один и тот же сигнал.

Возьмите эту модель и, например, сигнал

```
class Event(models.Model):
    user = models.ForeignKey(User)
```

```
class StatusChange(Event):
    ...


class Comment(Event):
    ...


def send_activity_notification(sender, instance: Event, raw: bool, **kwargs):
    """
    Fire a notification upon saving an event
    """

    if not raw:
        msg_factory = MessageFactory(instance.id)
        msg_factory.on_activity(str(instance))
post_save.connect(send_activity_notification, Event)
```

В расширенных моделях вы должны вручную присоединить сигнал к каждому подклассу, иначе они не будут выполняться.

```
post_save.connect(send_activity_notification, StatusChange)
post_save.connect(send_activity_notification, Comment)
```

С Python 3.6 вы можете использовать некоторые дополнительные методы класса, встроенные в классы, чтобы автоматизировать эту привязку.

```
class Event(models.Model):

    @classmethod
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        post_save.connect(send_activity_notification, cls)
```

Прочитайте сигналы онлайн: https://riptutorial.com/ru/django/topic/2555/сигналы

глава 46: Справочник по заданию модели

параметры

параметр	подробности
ноль	Если true, пустые значения могут быть сохранены как `null` в базе данных
пустой	Если true, тогда поле не потребуется в формах. Если поля оставлены пустыми, Django будет использовать значение поля по умолчанию.
выбор	Итерируемый двухэлементный итерабель, который будет использоваться в качестве выбора для этого поля. Если установлено, поле отображается как раскрывающееся меню администратора. `[('m', 'Male'),('f','Female'),('z','Prefer Not to Disclose')]` . Чтобы группировать параметры, просто вставьте значения: `[('Video Source',((1,'YouTube'),(2,'Facebook')),('Audio Source',((3, 'Soundcloud'),(4, 'Spotify'))]`
db_column	По умолчанию django использует имя поля для столбца базы данных. Используйте это, чтобы указать собственное имя
db_index	Если `True` , индекс будет создан в этом поле в базе данных
db_tablespace	Табличное пространство, используемое для индекса этого поля. *Это поле используется только в том случае, если движок базы данных поддерживает его, иначе его игнорируют* .
дефолт	Значение по умолчанию для этого поля. Может быть значением или вызываемым объектом. Для изменяемых значений по умолчанию (список, набор, словарь) вы **должны** использовать вызываемый. Из-за совместимости с миграциями вы не можете использовать lambdas.
редактируемые	Если `False` , это поле не отображается в администраторе модели или любом `ModelForm` . Значение по умолчанию - `True` .
Сообщения об ошибках	Используется для настройки сообщений об ошибках по умолчанию, отображаемых для этого поля. Значение представляет собой словарь, с ключами, представляющими ошибку, и значением, являющимся сообщением. Клавиши по умолчанию (для сообщений об ошибках) являются `null` , `blank` , `invalid` , `invalid_choice` , `unique` и

параметр	подробности
	`unique_for_date` ; дополнительные сообщения об ошибках могут быть определены пользовательскими полями.
help_text	Текст, который будет отображаться в поле, чтобы помочь пользователям. HTML разрешен.
on_delete	Когда объект, на который ссылается ForeignKey, удаляется, Django будет эмулировать поведение ограничения SQL, указанного аргументом on_delete. Это второй позиционный аргумент для полей `ForeignKey` и `OneToOneField` . Другие поля не имеют этого аргумента.
основной ключ	Если `True` , это поле будет первичным ключом. Django автоматически добавляет первичный ключ; поэтому это требуется только в том случае, если вы хотите создать настраиваемый первичный ключ. У вас может быть только один первичный ключ для каждой модели.
уникальный	Если `True` , ошибки возникают, если для этого поля введены повторяющиеся значения. Это ограничение на уровне базы данных, а не просто блок пользовательского интерфейса.
unique_for_date	Задайте значение `DateField` или `DateTimeField` , и ошибки будут подняты, если есть повторяющиеся значения *для той же даты или даты* .
unique_for_month	Подобно `unique_for_date` , кроме проверок, ограниченного для месяца.
unique_for_year	Подобно `unique_for_date` , кроме проверок, ограничивается годом.
verbose_name	Дружественное имя для поля, используемое django в разных местах (например, создание меток в форме администратора и модели).
валидаторы	Список валидаторов для этого поля.

замечания

- Вы можете написать свои собственные поля, если найдете это необходимым
- Вы можете переопределить функции базового класса модели, чаще всего функцию `save()`

Examples

Число полей

Приведены примеры числовых полей:

AutoField

Автоматически увеличивающееся целое число, обычно используемое для первичных ключей.

```
from django.db import models

class MyModel(models.Model):
    pk = models.AutoField()
```

> По умолчанию каждая модель получает поле первичного ключа (называемое `id`). Поэтому нет необходимости дублировать поле id в модели для первичного ключа.

BigIntegerField

Целочисленные номера фитингов от `-9223372036854775808` до `9223372036854775807` (`8 Bytes`).

```
from django.db import models

class MyModel(models.Model):
    number_of_seconds = models.BigIntegerField()
```

IntegerField

IntegerField используется для хранения целочисленных значений от -2147483648 до 2147483647 (`4 Bytes`).

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.IntegerField(default=0)
```

> параметр по `default` не является обязательным. Но полезно установить значение по умолчанию.

PositiveIntegerField

Как IntegerField, но должен быть либо положительным, либо нулевым (0). PositiveIntegerField используется для хранения целочисленных значений от 0 до 2147483647 (`4 Bytes`). Это может быть полезно в поле, которое должно быть семантически положительным. Например, если вы записываете продукты с калориями, это не должно

быть отрицательным. Это поле предотвратит отрицательные значения посредством его проверки.

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.PositiveIntegerField(default=0)
```

> параметр по `default` не является обязательным. Но полезно установить значение по умолчанию.

SmallIntegerField

SmallIntegerField используется для хранения целочисленных значений от -32768 до 32767 (`2 Bytes`). Это поле полезно для значений, а не для экстремумов.

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=255)
    temperature = models.SmallIntegerField(null=True)
```

PositiveSmallIntegerField

SmallIntegerField используется для хранения целочисленных значений от 0 до 32767 (`2 Bytes`). Точно так же, как SmallIntegerField, это поле полезно для значений, не столь высоких и должно быть семантически положительным. Например, он может хранить возраст, который не может быть отрицательным.

```
from django.db import models

class Staff(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    age = models.PositiveSmallIntegerField(null=True)
```

Кроме того, PositiveSmallIntegerField полезен для выбора, это способ Djangoic для реализации Enum:

```
from django.db import models
from django.utils.translation import gettext as _

APPLICATION_NEW = 1
APPLICATION_RECEIVED = 2
APPLICATION_APPROVED = 3
APPLICATION_REJECTED = 4

APLICATION_CHOICES = (
    (APPLICATION_NEW, _('New')),
```

```
    (APPLICATION_RECEIVED, _('Received')),
    (APPLICATION_APPROVED, _('Approved')),
    (APPLICATION_REJECTED, _('Rejected')),
)

class JobApplication(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    status = models.PositiveSmallIntegerField(
        choices=APLICATION_CHOICES,
        default=APPLICATION_NEW
    )
    ...
```

> Определение выбора в качестве переменных класса или переменных модуля в зависимости от ситуации - хороший способ их использования. Если выбор передается в поле без дружественных имен, это создаст путаницу.

DecimalField

Десятичное число с фиксированной точностью, представленное в Python экземпляром Decimal. В отличие от IntegerField и его производных это поле имеет 2 требуемых аргумента:

1. *DecimalField.max_digits* : максимальное количество цифр, разрешенных в номере. Обратите внимание, что это число должно быть больше или равно decimal_places.
2. *DecimalField.decimal_places* : количество десятичных знаков для хранения с номером.

Если вы хотите хранить номера до 99 с 3 десятичными знаками, вам нужно использовать `max_digits=5` и `decimal_places=3` :

```
class Place(models.Model):
    name = models.CharField(max_length=255)
    atmospheric_pressure = models.DecimalField(max_digits=5, decimal_places=3)
```

BinaryField

Это специализированное поле, используемое для хранения двоичных данных. Он принимает только **байты** . Данные основаны на базе64 при хранении.

Поскольку это хранение двоичных данных, это поле не может использоваться в фильтре.

```
from django.db import models

class MyModel(models.Model):
    my_binary_data = models.BinaryField()
```

CharField

CharField используется для хранения определенных длин текста. В приведенном ниже примере в поле можно сохранить до 128 символов текста. Ввод строки дольше, чем это приведет к повышению ошибки проверки.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=128, blank=True)
```

DateTimeField

DateTimeField используется для хранения значений времени.

```
class MyModel(models.Model):
    start_time = models.DateFimeField(null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)
```

`DateTimeField` имеет два необязательных параметра:

- `auto_now_add` устанавливает значение поля в текущее время и дату, когда объект создается.
- `auto_now` устанавливает значение поля в текущее время и время при каждом сохранении поля.

Эти параметры и параметр по `default` являются взаимоисключающими.

Иностранный ключ

ForeignKey поля используются для создания `many-to-one` отношениям между моделями. Не похоже, что большинство других полей требуют позиционных аргументов. Следующий пример демонстрирует отношение к машине и владельцу:

```
from django.db import models

class Person(models.Model):
    GENDER_FEMALE = 'F'
    GENDER_MALE = 'M'

    GENDER_CHOICES = (
        (GENDER_FEMALE, 'Female'),
        (GENDER_MALE, 'Male'),
    )

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
    age = models.SmallIntegerField()
```

```
class Car(model.Model)
    owner = models.ForeignKey('Person')
    plate = models.CharField(max_length=15)
    brand = models.CharField(max_length=50)
    model = models.CharField(max_length=50)
    color = models.CharField(max_length=50)
```

Первый аргумент поля - это класс, к которому относится модель. Второй позиционный аргумент - аргумент `on_delete` . В текущих версиях этот аргумент не требуется, но он потребуется в Django 2.0. Функциональность аргумента по умолчанию показана следующим образом:

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.CASCADE)
    ...
```

> Это приведет к удалению объектов Car из модели, когда ее владелец удалил из модели Person. Это функциональность по умолчанию.

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.PROTECT)
    ...
```

> Это предотвратит удаление объектов Person, если они связаны с хотя бы одним объектом Car. Все объекты Car, которые ссылаются на объект Person, должны быть удалены первыми. И тогда объект Person может быть удален.

Прочитайте Справочник по заданию модели онлайн:
https://riptutorial.com/ru/django/topic/3686/справочник-по-заданию-модели

глава 47: Структура проекта

Examples

Репозиторий> Проект> Сайт / Конф.

Для проекта Django с `requirements` и `deployment tools` под контролем источника. Этот пример основан на концепциях из двух совпадений Django . Они опубликовали шаблон :

```
repository/
    docs/
    .gitignore
    project/
        apps/
            blog/
                migrations/
                static/ #( optional )
                    blog/
                        some.css
                templates/ #( optional )
                    blog/
                        some.html
                models.py
                tests.py
                admin.py
                apps.py #( django 1.9 and later )
                views.py
            accounts/
                #... ( same as blog )
            search/
                #... ( same as blog )
        conf/
            settings/
                local.py
                development.py
                production.py
            wsgi
            urls.py
        static/
        templates/
    deploy/
        fabfile.py
    requirements/
        base.txt
        local.txt
    README
    AUTHORS
    LICENSE
```

Здесь `apps` и папки `conf` содержат `user created applications` и `core configuration folder` для проекта соответственно.

`static` и `templates` папки в каталоге `project` содержат статические файлы и файлы `html`

`markup` соответственно, которые используются во всем мире во всем мире.

И все `blog` приложений, `accounts` и `search` могут также (в основном) содержать `static` и `templates` папки.

Namespacing статические и шаблонные файлы в приложениях django

`static` и `templates` в приложениях может также содержать папку с именем приложения `ex. blog` это соглашение, используемое для предотвращения загрязнения пространства имен, поэтому мы `/blog/base.html` на файлы, такие как `/blog/base.html` а не на `/base.html` который обеспечивает большую ясность в отношении файла, на который мы `/base.html` и сохраняет пространство имен.

Пример: папка `templates` внутри `blog` и `search` приложений содержит файл с именем `base.html` , а при обращении к файлу в `views` ваше приложение запутывается в каком файле для рендеринга.

```
(Project Structure)
.../project/
    apps/
        blog/
            templates/
                base.html
        search/
            templates/
                base.html

(blog/views.py)
def some_func(request):
    return render(request, "/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/base.html")

## After creating a folder inside /blog/templates/(blog) ##

(Project Structure)
.../project/
    apps/
        blog/
            templates/
                blog/
                    base.html
        search/
            templates/
                search/
                    base.html

(blog/views.py)
def some_func(request):
    return render(request, "/blog/base.html")

(search/views.py)
def some_func(request):
```

```
    return render(request, "/search/base.html")
```

Прочитайте Структура проекта онлайн: https://riptutorial.com/ru/django/topic/4299/структура-проекта

глава 48: Теги шаблонов и фильтры

Examples

Пользовательские фильтры

Фильтры позволяют применять функцию к переменной. Эта функция может принимать **0** или **1** аргумент. Вот синтаксис:

```
{{ variable|filter_name }}
{{ variable|filter_name:argument }}
```

Фильтры могут быть скованы так, что это совершенно верно:

```
{{ variable|filter_name:argument|another_filter }}
```

Если перевести на python, указанная выше строка даст что-то вроде этого:

```
print(another_filter(filter_name(variable, argument)))
```

В этом примере мы напишем настраиваемый `verbose_name` который применяется к модели (экземпляру или классу) или QuerySet. Он вернет подробное имя модели или ее многословное имя, если для аргумента установлено значение `True` .

```
@register.filter
def verbose_name(model, plural=False):
    """Return the verbose name of a model.
    `model` can be either:
      - a Model class
      - a Model instance
      - a QuerySet
      - any object refering to a model through a `model` attribute.

    Usage:
      - Get the verbose name of an object
          {{ object|verbose_name }}
      - Get the plural verbose name of an object from a QuerySet
          {{ objects_list|verbose_name:True }}
    """
    if not hasattr(model, '_meta'):
        # handle the case of a QuerySet (among others)
        model = model.model
    opts = model._meta
    if plural:
        return opts.verbose_name_plural
    else:
        return opts.verbose_name
```

Простые теги

Самый простой способ определить собственный тег шаблона - использовать `simple_tag` . Это очень просто настроить. Имя функции будет именем тега (хотя вы можете переопределить его), а аргументы будут токенами («слова» разделены пробелами, кроме пробелов, заключенных между кавычками). Он даже поддерживает ключевые слова.

Вот бесполезный тег, который иллюстрирует наш пример:

```
{% useless 3 foo 'hello world' foo=True bar=baz.hello|capfirst %}
```

Пусть `foo` и `baz` являются контекстными переменными, такими как:

```
{'foo': "HELLO", 'baz': {'hello': "world"}}
```

Скажем, мы хотим, чтобы этот очень бесполезный тег отображался следующим образом:

```
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
```

Пример конкатенации аргументов повторяется 3 раза (3 - первый аргумент).

Вот как выглядит реализация тега:

```
from django.utils.html import format_html_join

@register.simple_tag
def useless(repeat, *args, **kwargs):
    output = ';'.join(args + ['{}:{}'.format(*item) for item in kwargs.items()])
    outputs = [output] * repeat
    return format_html_join('\n', '{}<br/>', ((e,) for e in outputs))
```

`format_html_join` позволяет отмечать `<br/>` как безопасный HTML, но не содержимое `outputs` .

Расширенные пользовательские теги с использованием узла

Иногда то, что вы хотите сделать, слишком сложно для `filter` или `simple_tag` . Для этого вам нужно создать функцию компиляции и средство визуализации.

В этом примере мы создадим тег шаблона `verbose_name` со следующим синтаксисом:

пример	Описание
`{% verbose_name obj %}`	Подробное имя модели
`{% verbose_name obj 'status' %}`	Подробное имя поля "status"
`{% verbose_name obj plural %}`	Многословное имя множественного числа модели

пример	Описание
`{% verbose_name obj plural capfirst %}`	Заглавное многословное имя множественного числа модели
`{% verbose_name obj 'foo' capfirst %}`	Расширенное подробное имя поля
`{% verbose_name obj field_name %}`	Подробное имя поля из переменной
`{% verbose_name obj 'foo'|add:'_bar' %}`	Подробное имя поля "foo_bar"

Причина, по которой мы не можем сделать это с помощью простого тега, состоит в том, что `plural` и `capfirst` являются ни переменными, ни строками, они являются «ключевыми словами». Очевидно, мы решили передать их как `'plural'` строк и `'capfirst'` , но они могут конфликтовать с полями с этими именами. Было бы `{% verbose_name obj 'plural' %}` означать "многословное имя множественного числа `obj` " или "подробное имя `obj.plural` "?

Сначала создадим функцию компиляции:

```
@register.tag(name='verbose_name')
def do_verbose_name(parser, token):
    """
    - parser: the Parser object. We will use it to parse tokens into
              nodes such as variables, strings, ...
    - token: the Token object. We will use it to iterate each token
             of the template tag.
    """
    # Split tokens within spaces (except spaces inside quotes)
    tokens = token.split_contents()
    tag_name = tokens[0]
    try:
        # each token is a string so we need to parse it to get the actual
        # variable instead of the variable name as a string.
        model = parser.compile_filter(tokens[1])
    except IndexError:
        raise TemplateSyntaxError(
            "'{}' tag requires at least 1 argument.".format(tag_name))

    field_name = None
    flags = {
        'plural': False,
        'capfirst': False,
    }

    bits = tokens[2:]
    for bit in bits:
        if bit in flags.keys():
            # here we don't need `parser.compile_filter` because we expect
            # 'plural' and 'capfirst' flags to be actual strings.
            if flags[bit]:
                raise TemplateSyntaxError(
                    "'{}' tag only accept one occurrence of '{}' flag".format(
                        tag_name, bit)
                )
            flags[bit] = True
```

```
            continue
        if field_name:
            raise TemplateSyntaxError((
                "'{}' tag only accept one field name at most. {} is the second "
                "field name encountered."
            ).format(tag_name, bit)
        field_name = parser.compile_filter(bit)

    # VerboseNameNode is our renderer which code is given right below
    return VerboseNameNode(model, field_name, **flags)
```

И теперь рендеринг:

```
class VerboseNameNode(Node):

    def __init__(self, model, field_name=None, **flags):
        self.model = model
        self.field_name = field_name
        self.plural = flags.get('plural', False)
        self.capfirst = flags.get('capfirst', False)

    def get_field_verbose_name(self):
        if self.plural:
            raise ValueError("Plural is not supported for fields verbose name.")
        return self.model._meta.get_field(self.field_name).verbose_name

    def get_model_verbose_name(self):
       if self.plural:
           return self.model._meta.verbose_name_plural
       else:
           return self.model._meta.verbose_name

    def render(self, context):
        """This is the main function, it will be called to render the tag.
        As you can see it takes context, but we don't need it here.
        For instance, an advanced version of this template tag could look for an
        `object` or `object_list` in the context if `self.model` is not provided.
        """
        if self.field_name:
            verbose_name = self.get_field_verbose_name()
        else:
            verbose_name = self.get_model_verbose_name()
        if self.capfirst:
            verbose_name = verbose_name.capitalize()
        return verbose_name
```

Прочитайте Теги шаблонов и фильтры онлайн: https://riptutorial.com/ru/django/topic/1305/теги-шаблонов-и-фильтры

глава 49: Тестирование устройства

Examples

Тестирование - полный пример

Предполагается, что вы прочитали документацию о запуске нового проекта Django. Предположим, что основное приложение в вашем проекте называется td (short для test driven). Чтобы создать свой первый тест, создайте файл с именем test_view.py и скопируйте в него следующий контент.

```
from django.test import Client, TestCase

class ViewTest(TestCase):

    def test_hello(self):
        c = Client()
        resp = c.get('/hello/')
        self.assertEqual(resp.status_code, 200)
```

Вы можете запустить этот тест

```
 ./manage.py test
```

и это, естественно, потерпит неудачу! Вы увидите ошибку, подобную следующей.

```
Traceback (most recent call last):
  File "/home/me/workspace/td/tests_view.py", line 9, in test_hello
    self.assertEqual(resp.status_code, 200)
AssertionError: 200 != 404
```

Почему это происходит? Потому что мы не определили для этого точку зрения! Так что давайте сделаем это. Создайте файл с именем views.py и поместите в него следующий код

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse('hello')
```

Затем переместите его в / hello /, отредактировав URL-адрес py следующим образом:

```
from td import views

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/', views.hello),
    ....
]
```

Теперь повторите тест снова. / `./manage.py test` снова и альта !!

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

Эффективное тестирование моделей Django

Предполагая класс

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('view_author', args=[str(self.id)])


class Book(models.Model):
    author = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    private = models.BooleanField(default=false)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('view_book', args=[str(self.id)])

    def __str__(self):
        return self.name
```

Примеры тестирования

```
from django.test import TestCase
from .models import Book, Author

class BaseModelTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseModelTestCase, cls).setUpClass()
        cls.author = Author(name='hawking')
        cls.author.save()
        cls.first_book = Book(author=cls.author, name="short_history_of_time")
        cls.first_book.save()
        cls.second_book = Book(author=cls.author, name="long_history_of_time")
        cls.second_book.save()


class AuthorModelTestCase(BaseModelTestCase):
    def test_created_properly(self):
         self.assertEqual(self.author.name, 'hawking')
```

```
         self.assertEqual(True, self.first_book in self.author.book_set.all())

    def test_absolute_url(self):
        self.assertEqual(self.author.get_absolute_url(), reverse('view_author',
args=[str(self.author.id)]))

class BookModelTestCase(BaseModelTestCase):

    def test_created_properly(self:
        ...
        self.assertEqual(1, len(Book.objects.filter(name__startswith='long'))

    def test_absolute_url(self):
        ...
```

Некоторые моменты

- Тесты `created_properly` используются для проверки свойств состояния моделей django. Они помогают поймать sitautions, где мы изменили значения по умолчанию, file_upload_paths и т. Д.
- `absolute_url` может показаться тривиальным, но я обнаружил, что это помогло мне предотвратить некоторые ошибки при изменении URL-адресов
- Я также записываю тестовые примеры для всех методов, реализованных внутри модели (с использованием `mock` объектов и т. Д.),
- Определяя общую `BaseModelTestCase` мы можем установить необходимые отношения между моделями для обеспечения правильного тестирования.

Наконец, когда вы сомневаетесь, напишите тест. Тривиальные изменения поведения улавливаются, обращая внимание на детали, и давно забытые фрагменты кода не приводят к ненужным проблемам.

Тестирование контроля доступа в представлениях Django

tl; dr : создать базовый класс, который определяет два пользовательских объекта (например, `user` и `another_user`). Создайте другие модели и определите три экземпляра `Client` .

- `self.client` : представление `user` зарегистрированного в браузере.
- `self.another_client` : представление клиента `another_user`
- `self.unlogged_client` : представление незарегистрированного лица

Теперь обращайтесь ко всем своим публичным и частным адресам из этих трех клиентских объектов и диктуйте ответ, который вы ожидаете. Ниже я продемонстрировал стратегию для объекта `Book` который может быть либо `private` (принадлежащим нескольким привилегированным пользователям), либо `public` (видимым для всех).

```
from django.test import TestCase, RequestFactory, Client
from django.core.urlresolvers import reverse
```

```
class BaseViewTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseViewTestCase, cls).setUpClass()
        cls.client = Client()
        cls.another_client = Client()
        cls.unlogged_client = Client()
        cls.user = User.objects.create_user(
                'dummy',password='dummy'
                )
        cls.user.save()
        cls.another_user = User.objects.create_user(
                'dummy2', password='dummy2'
                )
        cls.another_user.save()
        cls.first_book = Book.objects.create(
                name='first',
                private = true
        )
        cls.first_book.readers.add(cls.user)
        cls.first_book.save()
        cls.public_book = Template.objects.create(
                name='public',
                private=False
        )
        cls.public_book.save()


    def setUp(self):
        self.client.login(username=self.user.username, password=self.user.username)
        self.another_client.login(username=self.another_user.username,
password=self.another_user.username)


"""
   Only cls.user owns the first_book and thus only he should be able to see it.
   Others get 403(Forbidden) error
"""
class PrivateBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PrivateBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.first_book.id)})

    def test_user_sees_own_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.first_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_user_cant_see_others_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(403, response.status_code)

    def test_unlogged_user_cant_see_private_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(403, response.status_code)

"""
    Since book is public all three clients should be able to see the book
```

```
"""
 class PublicBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PublicBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.public_book.id)})

    def test_user_sees_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.public_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_another_user_sees_public_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(200, response.status_code)

    def test_unlogged_user_sees_public_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(200, response.status_code)
```

База данных и тестирование

Django использует специальные настройки базы данных при тестировании, чтобы тесты могли нормально использовать базу данных, но по умолчанию запускались в пустой базе данных. Изменение базы данных в одном тесте не будет видно другим. Например, оба следующих теста пройдут:

```
from django.test import TestCase
from myapp.models import Thing

class MyTest(TestCase):

    def test_1(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create()
        self.assertEqual(Thing.objects.count(), 1)

    def test_2(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create(attr1="value")
        self.assertEqual(Thing.objects.count(), 1)
```

арматура

Если вы хотите, чтобы объекты базы данных использовали несколько тестов, либо создайте их в методе `setUp` тестового примера. Кроме того, если вы определили приборы в своем проекте django, их можно включить так:

```
class MyTest(TestCase):
    fixtures = ["fixture1.json", "fixture2.json"]
```

По умолчанию django ищет светильники в каталоге `fixtures` в каждом приложении. Дальнейшие каталоги можно установить с помощью параметра `FIXTURE_DIRS` :

```
# myapp/settings.py
FIXTURE_DIRS = [
    os.path.join(BASE_DIR, 'path', 'to', 'directory'),
]
```

Предположим, вы создали модель следующим образом:

```
# models.py
from django.db import models


class Person(models.Model):
    """A person defined by his/her first- and lastname."""
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
```

Тогда ваши .json светильники могли бы выглядеть так:

```
# fixture1.json
[
    { "model": "myapp.person",
        "pk": 1,
        "fields": {
            "firstname": "Peter",
            "lastname": "Griffin"
        }
    },
    { "model": "myapp.person",
        "pk": 2,
        "fields": {
            "firstname": "Louis",
            "lastname": "Griffin"
        }
    },
]
```

Повторное использование тестовой базы данных

Чтобы ускорить ваши тестовые тесты, вы можете сообщить команде управления повторно использовать тестовую базу данных (и предотвратить ее создание до и удалить после каждого теста). Это можно сделать, используя флаг keepdb (или стенограммы `-k`):

```
# Reuse the test-database (since django version 1.8)
$ python manage.py test --keepdb
```

Ограничьте количество выполненных тестов

Можно ограничить тесты, выполненные с помощью `manage.py test` , указав, какие модули должны быть обнаружены тестовым бегуном:

```
# Run only tests for the app names "app1"
$ python manage.py test app1
```

```
# If you split the tests file into a module with several tests files for an app
$ python manage.py test app1.tests.test_models

# it's possible to dig down to individual test methods.
$ python manage.py test app1.tests.test_models.MyTestCase.test_something
```

Если вы хотите запустить кучу тестов, вы можете передать шаблон имен файлов. Например, вы можете запускать только те тесты, которые связаны с вашими моделями:

```
$ python manage.py test -p test_models*
Creating test database for alias 'default'...
.............................................
----------------------------------------------------------------------
Ran 115 tests in 3.869s

OK
```

Наконец, можно остановить набор тестов при первом `--failfast` , используя `--failfast` . Этот аргумент позволяет быстро получить потенциальную ошибку, встречающуюся в пакете:

```
$ python manage.py test app1
...F..
----------------------------------------------------------------------
Ran 6 tests in 0.977s

FAILED (failures=1)


$ python manage.py test app1 --failfast
...F
======================================================================
[Traceback of the failing test]
----------------------------------------------------------------------
Ran 4 tests in 0.372s

FAILED (failures=1)
```

Прочитайте Тестирование устройства онлайн: https://riptutorial.com/ru/django/topic/1232/тестирование-устройства

глава 50: Транзакции базы данных

Examples

Атомные транзакции

проблема

По умолчанию Django немедленно вносит изменения в базу данных. Когда исключения происходят во время серии коммитов, это может оставить вашу базу данных в нежелательном состоянии:

```
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

В следующем сценарии:

```
>>> create_category('clothing', ['shirt', 'trousers', 'tie'])
--------------------------------------------------------------------------
ValueError: Product 'trousers' already exists
```

Исключение возникает при попытке добавить продукт брюк в категорию одежды. К этому моменту уже добавлена сама категория, и к ней добавлен продукт рубашки.

Неполную категорию и содержащий продукт нужно удалить вручную до исправления кода и вызова `create_category()` еще раз, так как в противном случае будет создана повторяющаяся категория.

Решение

Модуль `django.db.transaction` позволяет объединять несколько изменений базы данных в атомную транзакцию :

> [a] ряд операций с базой данных, в которых либо все происходит, либо ничего не происходит.

Применительно к описанному выше сценарию это можно применить в качестве декоратора :

```
from django.db import transaction

@transaction.atomic
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

Или с помощью диспетчера контекстов :

```
def create_category(name, products):
    with transaction.atomic():
        category = Category.objects.create(name=name)
        product_api.add_products_to_category(category, products)
        activate_category(category)
```

Теперь, если исключение происходит на любом этапе транзакции, никаких изменений базы данных не будет.

Прочитайте Транзакции базы данных онлайн: https://riptutorial.com/ru/django/topic/5555/транзакции-базы-данных

глава 51: формы

Examples

Пример модели

Создайте ModelForm из существующего класса Model, подклассифицируя `ModelForm` :

```
from django import forms

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['item', 'order_date', 'customer', 'status']
```

Определение формы Django с нуля (с виджетами)

Формы могут быть определены аналогично моделям путем подкласса `django.forms.Form` . Доступны различные варианты ввода полей, такие как `CharField` , `URLField` , `IntegerField` и т. Д.

Определение простой формы контакта можно увидеть ниже:

```
from django import forms

class ContactForm(forms.Form):
    contact_name = forms.CharField(
        label="Your name", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    contact_email = forms.EmailField(
        label="Your Email Address", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    content = forms.CharField(
        label="Your Message", required=True,
        widget=forms.Textarea(attrs={'class': 'form-control'}))
```

Виджет представляет собой Django представление HTML-тегов, вводимых пользователем, и может использоваться для визуализации настраиваемого html для полей формы (например: как текстовое поле отображается для ввода содержимого здесь)

`attrs` - атрибуты, которые будут скопированы так же, как и отображаемый html для формы.

Например: `content.render("name", "Your Name")` дает

```
<input title="Your name" type="text" name="name" value="Your Name" class="form-control" />
```

Удаление поля modelForm на основе условия из views.py

Если у нас есть Модель как следующая,

```
from django.db import models
from django.contrib.auth.models import User

class UserModuleProfile(models.Model):
    user = models.OneToOneField(User)
    expired = models.DateTimeField()
    admin = models.BooleanField(default=False)
    employee_id = models.CharField(max_length=50)
    organisation_name = models.ForeignKey('Organizations', on_delete=models.PROTECT)
    country = models.CharField(max_length=100)
    position = models.CharField(max_length=100)

    def __str__(self):
        return self.user
```

И модельная форма, которая использует эту модель как следующую,

```
from .models import UserModuleProfile, from django.contrib.auth.models import User
from django import forms

class UserProfileForm(forms.ModelForm):
    admin = forms.BooleanField(label="Make this User
Admin",widget=forms.CheckboxInput(),required=False)
    employee_id = forms.CharField(label="Employee Id ")
    organisation_name = forms.ModelChoiceField(label='Organisation
Name',required=True,queryset=Organizations.objects.all(),empty_label="Select an Organization")
    country = forms.CharField(label="Country")
    position = forms.CharField(label="Position")

    class Meta:
        model = UserModuleProfile
        fields = ('admin','employee_id','organisation_name','country','position',)

    def __init__(self, *args, **kwargs):
        admin_check = kwargs.pop('admin_check', False)
        super(UserProfileForm, self).__init__(*args, **kwargs)
        if not admin_check:
            del self.fields['admin']
```

Обратите внимание, что ниже класса Meta в форме я добавил функцию **init,** которую мы можем использовать при инициализации формы из views.py, чтобы удалить поле формы (или некоторые другие действия). Я объясню это позже.

Таким образом, эта форма может использоваться для целей регистрации пользователей, и мы хотим, чтобы все поля были определены в мета-классе формы. Но что, если мы хотим использовать ту же форму при редактировании пользователя, но когда мы это делаем, мы не хотим показывать поле admin формы?

Мы можем просто отправить дополнительный аргумент, когда мы инициализируем форму на основе некоторой логики и удаляем поле admin из бэкэнд.

```
def edit_profile(request,user_id):
```

```
    context = RequestContext(request)
    user = get_object_or_404(User, id=user_id)
    profile = get_object_or_404(UserModuleProfile, user_id=user_id)
    admin_check = False
    if request.user.is_superuser:
        admin_check = True
    # If it's a HTTP POST, we're interested in processing form data.
    if request.method == 'POST':
        # Attempt to grab information from the raw form information.
        profile_form =
UserProfileForm(data=request.POST,instance=profile,admin_check=admin_check)
        # If the form is valid...
        if profile_form.is_valid():
            form_bool = request.POST.get("admin", "xxx")
            if form_bool == "xxx":
                form_bool_value = False
            else:
                form_bool_value = True
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.admin = form_bool_value
            profile.save()
            edited = True
        else:
            print profile_form.errors

    # Not a HTTP POST, so we render our form using ModelForm instance.
    # These forms will be blank, ready for user input.
    else:
        profile_form = UserProfileForm(instance = profile,admin_check=admin_check)

    return render_to_response(
            'usermodule/edit_user.html',
            {'id':user_id, 'profile_form': profile_form, 'edited': edited, 'user':user},
            context)
```

Как вы можете видеть, я показал здесь простой пример редактирования, используя форму, которую мы создали ранее. Обратите внимание, когда я инициализировал форму, я передал дополнительную переменную `admin_check` которая содержит либо `True` либо `False` .

```
profile_form = UserProfileForm(instance = profile,admin_check=admin_check)
```

Теперь, если вы заметили форму, которую мы написали ранее, вы можете увидеть, что в **init** мы пытаемся поймать параметр `admin_check` который мы передаем отсюда. Если значение False, мы просто удаляем поле `admin` из формы и используем его. И так как это модельное поле для администратора формы не может быть пустым в модели, мы просто проверяем, было ли в поле формы поле admin в сообщении формы, если мы не установили его в `False` в коде вида в следующем коде представления.

```
form_bool = request.POST.get("admin", "xxx")
if form_bool == "xxx":
    form_bool_value = False
else:
    form_bool_value = True
```

Загрузка файлов с помощью форм Django

Прежде всего, нам нужно добавить `MEDIA_ROOT` и `MEDIA_URL` в наш файл `settings.py`

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Также здесь вы будете работать с `ImageField` , так что помните в таких случаях, установите Pillow library (`pip install pillow`). В противном случае у вас будет такая ошибка:

```
ImportError: No module named PIL
```

Подушка - это вилка PIL, Python Imaging Library, которая больше не поддерживается. Подушка обратно совместима с PIL.

Django поставляется с двумя полями формы для загрузки файлов на сервер, `FileField` и `ImageField` , следующий пример использования этих двух полей в нашей форме

forms.py:

```
from django import forms


class UploadDocumentForm(forms.Form):
    file = forms.FileField()
    image = forms.ImageField()
```

views.py:

```
from django.shortcuts import render
from .forms import UploadDocumentForm


def upload_doc(request):
    form = UploadDocumentForm()
    if request.method == 'POST':
        form = UploadDocumentForm(request.POST, request.FILES)  # Do not forget to add:
request.FILES
        if form.is_valid():
            # Do something with our files or simply save them
            # if saved, our files would be located in media/ folder under the project's base
folder
            form.save()
    return render(request, 'upload_doc.html', locals())
```

upload_doc.html:

```
<html>
    <head>File Uploads</head>
    <body>
        <form enctype="multipart/form-data" action="" method="post"> <!-- Do not forget to
add: enctype="multipart/form-data" -->
```

```
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Save">
        </form>
    </body>
</html>
```

Проверка полей и Commit to model (Изменение электронной почты пользователя)

В Django уже реализованы формы для изменения пароля пользователя, одним из которых является SetPasswordForm .

Однако не существует форм для изменения электронной почты пользователя, и я думаю, что следующий пример очень важен, чтобы понять, как правильно использовать форму.

Следующий пример выполняет следующие проверки:

- Электронная почта на самом деле изменилась - очень полезно, если вам нужно проверить электронную почту или обновить шимпанзе;
- Как по электронной почте, так и по электронной почте с подтверждением совпадают - у формы есть два поля для электронной почты, поэтому обновление меньше подвержено ошибкам.

И в итоге он сохраняет новое электронное письмо в пользовательском объекте (обновляет электронную почту пользователя). Обратите внимание, что для метода `__init__()` требуется объект пользователя.

```
class EmailChangeForm(forms.Form):
    """
    A form that lets a user change set their email while checking for a change in the
    e-mail.
    """
    error_messages = {
        'email_mismatch': _("The two email addresses fields didn't match."),
        'not_changed': _("The email address is the same as the one already defined."),
    }

    new_email1 = forms.EmailField(
        label=_("New email address"),
        widget=forms.EmailInput,
    )

    new_email2 = forms.EmailField(
        label=_("New email address confirmation"),
        widget=forms.EmailInput,
    )

    def __init__(self, user, *args, **kwargs):
        self.user = user
        super(EmailChangeForm, self).__init__(*args, **kwargs)

    def clean_new_email1(self):
```

```
        old_email = self.user.email
        new_email1 = self.cleaned_data.get('new_email1')
        if new_email1 and old_email:
            if new_email1 == old_email:
                raise forms.ValidationError(
                    self.error_messages['not_changed'],
                    code='not_changed',
                )
        return new_email1

    def clean_new_email2(self):
        new_email1 = self.cleaned_data.get('new_email1')
        new_email2 = self.cleaned_data.get('new_email2')
        if new_email1 and new_email2:
            if new_email1 != new_email2:
                raise forms.ValidationError(
                    self.error_messages['email_mismatch'],
                    code='email_mismatch',
                )
        return new_email2

    def save(self, commit=True):
        email = self.cleaned_data["new_email1"]
        self.user.email = email
        if commit:
            self.user.save()
        return self.user



def email_change(request):
    form = EmailChangeForm()
    if request.method=='POST':
        form = Email_Change_Form(user,request.POST)
        if form.is_valid():
            if request.user.is_authenticated:
                if form.cleaned_data['email1']  == form.cleaned_data['email2']:
                    user = request.user
                    u = User.objects.get(username=user)
                    # get the proper user
                    u.email = form.cleaned_data['email1']
                    u.save()
                    return HttpResponseRedirect("/accounts/profile/")
    else:
        return render_to_response("email_change.html", {'form':form},
                                  context_instance=RequestContext(request))
```

Прочитайте формы онлайн: https://riptutorial.com/ru/django/topic/1217/формы

глава 52: Часовые пояса

Вступление

Часовые пояса часто являются препятствием для разработчиков. Django предлагает вам отличные утилиты, чтобы упростить работу с часовыми поясами.

Даже если ваш проект работает в одном часовом поясе, по-прежнему хорошей практикой является хранение данных в формате UTC в вашей базе данных для обработки случаев экономии дневного света. Если вы работаете в нескольких часовых поясах, то сохранение данных времени в формате UTC является обязательным.

Examples

Включить поддержку часового пояса

Во-первых, убедитесь, что `USE_TZ = True` в файле `settings.py` . Также установите значение по часовой `TIME_ZONE` по умолчанию `TIME_ZONE` например `TIME_ZONE='UTC'` . Просмотрите полный список часовых поясов здесь .

Если `USE_TZ` является False, `TIME_ZONE` будет часовым поясом, который Django будет использовать для хранения всех дат. Когда `USE_TZ` включен, `TIME_ZONE` является часовым поясом по умолчанию, который Django будет использовать для отображения данных в шаблонах и для интерпретации дат, введенных в формы.

При включенной поддержке часового пояса, Джанго будет хранить `datetime` - `UTC datetime` данных в базе данных в часовом поясе `UTC`

Установка часовых поясов сеанса

У объектов Python `datetime.datetime` есть атрибут `tzinfo` который используется для хранения информации о часовом поясе. Когда атрибут установлен, объект считается Aware, когда атрибут не установлен, он считается наивным.

Чтобы гарантировать, что часовой пояс наивен или осведомлен, вы можете использовать `.is_naive()` и `.is_aware()`

Если `USE_TZ` включены в вашем `settings.py` файла, то `datetime` и `TIME_ZONE settings.py datetime` будет иметь информацию о время зоны прикреплена к нему до тех пор , как ваш умолчанию `TIME_ZONE` установлен в `settings.py`

Хотя этот часовой пояс по умолчанию может быть хорошим, в некоторых случаях это, вероятно, недостаточно, особенно если вы обрабатываете пользователей в нескольких

часовых поясах. Для этого необходимо использовать промежуточное программное обеспечение.

```
import pytz

from django.utils import timezone

# make sure you add `TimezoneMiddleware` appropriately in settings.py
class TimezoneMiddleware(object):
    """
    Middleware to properly handle the users timezone
    """

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # make sure they are authenticated so we know we have their tz info.
        if request.user.is_authenticated():
            # we are getting the users timezone that in this case is stored in
            # a user's profile
            tz_str = request.user.profile.timezone
            timezone.activate(pytz.timezone(tz_str))
        # otherwise deactivate and the default time zone will be used anyway
        else:
            timezone.deactivate()

        response = self.get_response(request)
        return response
```

Происходит несколько новых вещей. Чтобы узнать больше о промежуточном программном обеспечении и о том, что он делает, проверьте эту часть документации . В `__call__` мы обрабатываем настройку данных часового пояса. Сначала мы проверяем подлинность пользователя, чтобы убедиться, что у нас есть данные о часовом поясе для этого пользователя. Как только мы узнаем, что мы делаем, мы активируем часовой пояс для сеанса пользователей, используя `timezone.activate()` . Чтобы преобразовать строку часового пояса, нам нужно что-то использовать в datetime, мы используем `pytz.timezone(str)` .

Теперь, когда объекты datetime доступны в шаблонах, они будут автоматически преобразованы из формата UTC базы данных в любой часовой пояс, в котором находится пользователь. Просто войдите в объект datetime, и его часовой пояс будет установлен, если будет установлено предыдущее промежуточное программное обеспечение должным образом.

```
{{ my_datetime_value }}
```

Если вы хотите получить мелкий контроль над тем, используется ли часовой пояс пользователя, посмотрите на следующее:

```
{% load tz %}
{% localtime on %}
```

```
    {# this time will be respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}

{% localtime off %}
    {# this will not respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}
```

Обратите внимание: этот метод описан только в Django 1.10 и on. Для поддержки django от 1.10 смотрите MiddlewareMixin

Прочитайте Часовые пояса онлайн: https://riptutorial.com/ru/django/topic/10566/часовые-пояса

глава 53: шаблонирование

Examples

переменные

Переменные, которые вы предоставили в контексте просмотра, можно получить с помощью двухстрочной записи:

На вашем `views.py` :

```
class UserView(TemplateView):
  """ Supply the request user object to the template """

  template_name = "user.html"

  def get_context_data(self, **kwargs):
    context = super(UserView, self).get_context_data(**kwargs)
    context.update(user=self.request.user)
    return context
```

В `user.html` :

```
<h1>{{ user.username }}</h1>

<div class="email">{{ user.email }}</div>
```

Точечная нотация будет доступна:

- свойства объекта, например `user.username` будет `{{ user.username }}`
- словарный поиск, например `request.GET["search"]` будет `{{ request.GET.search }}`
- методы без аргументов, например `users.count()` будет `{{ user.count }}`

Переменные шаблона не могут обращаться к методам, которые принимают аргументы.

Переменные также могут быть протестированы и зациклены:

```
{% if user.is_authenticated %}
  {% for item in menu %}
    <li><a href="{{ item.url }}">{{ item.name }}</a></li>
  {% endfor %}
{% else %}
  <li><a href="{% url 'login' %}">Login</a>
{% endif %}
```

Доступ к URL-адресам осуществляется с использованием формата `{% url 'name' %}` , где имена соответствуют именам в вашем `urls.py`

```
{% url 'login' %}
```

- вероятно, будет отображаться как `/accounts/login/`
`{% url 'user_profile' user.id %}` - Аргументы для URL-адресов поставляются в порядке
`{% url next %}` - URL-адреса могут быть переменными

Шаблоны в классах

Вы можете передавать данные в шаблон в пользовательской переменной.

На вашем `views.py` :

```
from django.views.generic import TemplateView
from MyProject.myapp.models import Item

class ItemView(TemplateView):
    template_name = "item.html"

    def items(self):
        """ Get all Items """
        return Item.objects.all()

    def certain_items(self):
        """ Get certain Items """
        return Item.objects.filter(model_field="certain")

    def categories(self):
        """ Get categories related to this Item """
        return Item.objects.get(slug=self.kwargs['slug']).categories.all()
```

Простой список в `item.html` :

```
{% for item in view.items %}
<ul>
    <li>{{ item }}</li>
</ul>
{% endfor %}
```

Вы также можете получить дополнительные свойства данных.

Если предположить , что модель `Item` имеет `name` поля:

```
{% for item in view.certain_items %}
<ul>
    <li>{{ item.name }}</li>
</ul>
{% endfor %}
```

Шаблоны в функциональных представлениях

Вы можете использовать шаблон в представлении, основанном на функции, следующим образом:

```
from django.shortcuts import render
```

```
def view(request):
    return render(request, "template.html")
```

Если вы хотите использовать переменные шаблона, вы можете сделать это следующим образом:

```
from django.shortcuts import render

def view(request):
    context = {"var1": True, "var2": "foo"}
    return render(request, "template.html", context=context)
```

Затем в `template.html` вы можете ссылаться на свои переменные следующим образом:

```
<html>
{% if var1 %}
    <h1>{{ var2 }}</h1>
{% endif %}
</html>
```

Фильтры шаблонов

Шаблонная система Django имеет встроенные *теги* и *фильтры* , которые являются функциями внутри шаблона для отображения контента определенным образом. Несколько фильтров могут быть указаны с помощью труб, а фильтры могут иметь аргументы, как и в переменном синтаксисе.

```
{{ "MAINROAD 3222"|lower }}    # mainroad 3222
{{ 10|add:15}}                 # 25
{{ "super"|add:"glue" }}       # superglue
{{ "A7"|add:"00" }}            # A700
{{ myDate | date:"D d M Y"}}   # Wed 20 Jul 2016
```

Список доступных **встроенных фильтров** можно найти на странице https://docs.djangoproject.com/en/dev/ref/templates/builtins/#ref-templates-builtins-filters .

Создание настраиваемых фильтров

Чтобы добавить собственные фильтры шаблонов, создайте в папке приложения папку с именем `templatetags` . Затем добавьте `__init__.py` и файл, в который будут входить фильтры:

```
#/myapp/templatetags/filters.py
from django import template

register = template.Library()

@register.filter(name='tostring')
def to_string(value):
    return str(value)
```

Чтобы использовать фильтр, вам необходимо загрузить его в свой шаблон:

```
#templates/mytemplate.html
{% load filters %}
{% if customer_id|tostring = customer %} Welcome back {% endif%}
```

Трюки

Несмотря на то, что фильтры сначала кажутся простыми, это позволяет сделать некоторые изящные вещи:

```
{% for x in ""|ljust:"20" %}Hello World!{% endfor %}    # Hello World!Hello World!Hel...
{{ user.name.split|join:"_" }} ## replaces whitespace with '_'
```

См. Также теги шаблонов для получения дополнительной информации.

Предотвращение вызова чувствительных методов в шаблонах

Когда объект подвергается контексту шаблона, доступны его методы без аргументов. Это полезно, когда эти функции являются «getters». Но это может быть опасно, если эти методы изменяют некоторые данные или имеют некоторые побочные эффекты. Несмотря на то, что вы, вероятно, доверяете создателю шаблона, он может не знать о побочных эффектах функции или ошибочно считать неправильный атрибут.

Учитывая следующую модель:

```
class Foobar(models.Model):
    points_credit = models.IntegerField()

    def credit_points(self, nb_points=1):
        """Credit points and return the new points credit value."""
        self.points_credit = F('points_credit') + nb_points
        self.save(update_fields=['points_credit'])
        return self.points_credit
```

Если вы напишете это, по ошибке, в шаблоне:

```
 You have {{ foobar.credit_points }} points!
```

Это будет увеличивать количество точек каждый раз при вызове шаблона. И вы даже этого не заметите.

Чтобы предотвратить это, вы должны установить для атрибута `alters_data` значение `True` для методов, имеющих побочные эффекты. Это сделает невозможным вызывать их из шаблона.

```
def credit_points(self, nb_points=1):
    """Credit points and return the new points credit value."""
    self.points_credit = F('points_credit') + nb_points
```

```
    self.save(update_fields=['points_credit'])
    return self.points_credit
credit_points.alters_data = True
```

Использование {% extends%}, {% include%} и {% blocks%}

резюме

- **{% extends%}** : объявляет шаблон, указанный в качестве аргумента как родитель текущего шаблона. Использование: `{% extends 'parent_template.html' %}` .
- **{% block%} {% endblock%}** : используется для определения разделов в ваших шаблонах, так что, если другой шаблон расширяет этот, он сможет заменить любой HTML-код, который был написан внутри него. Блоки идентифицируются по их имени. Использование: `{% block content %} <html_code> {% endblock %}` .
- **{% include%}** : это вставляет шаблон в текущий. Имейте в виду, что включенный шаблон получит контекст запроса, и вы также можете указать его собственные переменные. Основное использование: `{% include 'template_name.html' %}` , использование с переменными: `{% include 'template_name.html' with variable='value' variable2=8 %}`

Руководство

Предположим, что вы создаете код своей стороны, имея общие макеты для всего кода, и вы не хотите повторять код для каждого шаблона. Django дает вам встроенные теги для этого.
Предположим, у нас есть один блог-сайт с 3 шаблонами, которые имеют один и тот же макет:

```
project_directory
    ..
    templates
      front-page.html
      blogs.html
      blog-detail.html
```

1) Определите файл `base.html` ,

```
<html>
  <head>
  </head>

  <body>
        {% block content %}
        {% endblock %}
```

```
   </body>
</html>
```

2) Расширьте его в `blog.html` например,

```
{% extends 'base.html' %}

{% block content %}
    # write your blog related code here
{% endblock %}

# None of the code written here will be added to the template
```

Здесь мы расширили базовый макет, поэтому его макет HTML теперь доступен в `blog.html` blog.html. Концепция `{ % block %}` - это наследование шаблона, которое позволяет вам создать базовый шаблон скелета, который содержит все общие элементы вашего сайта и определяет блоки, которые дочерние шаблоны могут переопределить.

3) Теперь предположим, что все ваши 3 шаблона также имеют одинаковый HTML-div, который определяет некоторые популярные сообщения. Вместо того, чтобы писать три раза, создайте один новый шаблон `posts.html` .

blog.html

```
{% extends 'base.html' %}

{% block content %}
    # write your blog related code here
    {% include 'posts.html' %} # includes posts.html in blog.html file without passing any
data
    <!-- or -->
    {% include 'posts.html' with posts=postdata %} # includes posts.html in blog.html file
with passing posts data which is context of view function returns.
{% endblock %}
```

Прочитайте шаблонирование онлайн: https://riptutorial.com/ru/django/topic/588/шаблонирование

кредиты

S. No	Главы	Contributors
1	Начало работы с Django	A. Raza, Abhishek Jain, Aidas Bendoraitis, Alexander Tyapkov, Ankur Gupta, Anthony Pham, Antoine Pinsard, arifin4web, Community, e4c5, elbear, ericdwang, ettanany, Franck Dernoncourt, greatwolf, ilse2005, Ivan Semochkin, J F , Jared Hooper, John, John Moutafis, JRodDynamite, Kid Binary, knbk, Louis, Luis Alberto Santana, lxer, maciek, McAbra, MiniGunnR, mnoronha, Nathan Osman, naveen.panwar, nhydock, Nikita Davidenko, noɥʎdʎzɐɹƆ, Rahul Gupta, rajarshig, Ron, ruddra, sarvajeetsuman, shacker, ssice, Stryker, techydesigner, The Brewmaster, Thereissoupinmyfly, Tom, WesleyJohnson, Zags
2	ArrayField - поле PostgreSQL	Antoine Pinsard, e4c5, noɥʎdʎzɐɹƆ
3	CRUD в Django	aisflat439, George H.
4	Django Rest Framework	The Brewmaster
5	Django и социальные сети	Aidas Bendoraitis, aisflat439, Carlos Rojas, Ivan Semochkin, Rexford, Simplans
6	Django из командной строки.	e4c5, OliPro007
7	FormSets	naveen.panwar
8	JSONField - поле PostgreSQL	Antoine Pinsard, Daniil Ryzhkov, Matthew Schinckel, noɥʎdʎzɐɹƆ, Omar Shehata, techydesigner
9	Meta: Руководство по документации	Antoine Pinsard
10	Querysets	Antoine Pinsard, Brian Artschwager, Chalist, coffee-grinder, DataSwede, e4c5, Evans Murithi, George H., John Moutafis, Justin, knbk, Louis Barranqueiro, Maxime Lorant, MicroPyramid, nima, ravigadila, Sanyam Khurana, The Brewmaster
11	RangeFields - группа	Antoine Pinsard, noɥʎdʎzɐɹƆ

	полей PostgreSQL	
12	администрация	Antoine Pinsard, coffee-grinder, George H., Ivan Semochkin, noɥʎdʎzɐɹƆ, ssice
13	Асинхронные задачи (сельдерей)	iankit, Mevin Babu
14	Аутентификация	knbk, Rahul Gupta
15	Безопасность	Antoine Pinsard, knbk
16	Виджеты форм	Antoine Pinsard, ettanany
17	Выражения F ()	Antoine Pinsard, John Moutafis, Linville, Omar Shehata, RamenChef, Roald Nefs
18	Джанго-фильтр	4444, Ahmed Atalla
19	Запуск сельдерея с супервизором	RéÑjïth, sebb
20	интернационализация	Antoine Pinsard, dmvrtx
21	Использование Redis с Django - Caching Backend	Majid, The Brewmaster
22	Как восстановить миграцию django	Cristus Cleetus
23	Как использовать Django с Cookiecutter?	Atul Mishra, noɥʎdʎzɐɹƆ, OliPro007, RamenChef
24	Команды управления	Antoine Pinsard, aquasan, Brian Artschwager, HorsePunchKid, Ivan Semochkin, John Moutafis, knbk, lxer, MarZab, Nikolay Fominyh, pbaranay, ptim, Rana Ahmed, techydesigner, Zags
25	Контекстные процессоры	Antoine Pinsard, Brian Artschwager, Dan Russell, Daniil Ryzhkov, fredley
26	логирование	Antwane, Brian Artschwager, RamenChef
27	Маршрутизаторы баз данных	fredley, knbk
28	Маршрутизация URL	knbk

29	Миграции	Antoine Pinsard, engineercoding, Joey Wilhelm, knbk, MicroPyramid, ravigadila, Roald Nefs
30	модели	Aidas Bendoraitis, Alireza Aghamohammadi, alonisser, Antoine Pinsard, aquasan, Arpit Solanki, atomh33ls, coffee-grinder, DataSwede, ettanany, Gahan, George H., gkr, Ivan Semochkin, Jamie Cockburn, Joey Wilhelm, kcrk, knbk, Linville, lxer, maazza, Matt Seymour, MuYi, Navid777, nhydock, noɥʎdʎzɐɹƆ, pbaranay, PhoebeB, Rana Ahmed, Saksow, Sanyam Khurana, scriptmonster, Selcuk, SpiXel, sudshekhar, techydesigner, The_Cthulhu_Kid, Utsav T, waterproof, zurfyx
31	Модельные агрегации	Ian Clark, John Moutafis, ravigadila
32	Настройка базы данных	Ahmad Anwar, Antoine Pinsard, Evans Murithi, Kid Binary, knbk, lxer, Majid, Peter Mortensen
33	настройки	allo, Antoine Pinsard, Brian Artschwager, fredley, J F, knbk, Louis, Louis Barranqueiro, lxer, Maxime Lorant, NBajanca, Nils Werner, ProfSmiles, RamenChef, Sanyam Khurana, Sayse, Selcuk, SpiXel, ssice, sudshekhar, Tema, The Brewmaster
34	Непрерывная интеграция с Дженкинсом	pnovotnak
35	Общий вид	nikolas-berlin
36	отладка	Antoine Pinsard, Ashutosh, e4c5, Kid Binary, knbk, Sayse, Udi
37	Отношение «многие ко многим»	Antoine Pinsard, e4c5, knbk, Kostronor
38	Отображение строк в строки с помощью HStoreField - поле PostgreSQL	noɥʎdʎzɐɹƆ
39	Пользовательские менеджеры и запросы	abidibo, knbk, sudshekhar, Trivial
40	Представления на основе классов	Antoine Pinsard, Antwane, coffee-grinder, e4c5, gkr, knbk, maciek, masnun, Maxime Lorant, nicorellius,

		pleasedontbelong, Pureferret
41	Промежуточное	AlvaroAV, Antoine Pinsard, George H., knbk, lxer, nhydock, Omar Shehata, Peter Mortensen, Trivial, William Reed
42	Просмотры	ettanany, HorsePunchKid, John Moutafis
43	развертывание	Antoine Pinsard, Arpit Solanki, CodeFanatic23, I Am Batman, Ivan Semochkin, knbk, lxer, Maxime S., MaxLunar, Meska, no ɥʎdʎzɐɹƆ, rajarshig, Rishabh Agrahari, Roald Nefs, Rohini Choudhary, sebb
44	Расширение или замена модели пользователя	Antoine Pinsard, Jon Clements, mnoronha, Raito, Rexford, rigdonmr, Rishabh Agrahari, Roald Nefs, techydesigner, The_Cthulhu_Kid
45	сигналы	Antoine Pinsard, e4c5, Hetdev, John Moutafis, Majid, nhydock, Rexford
46	Справочник по заданию модели	Burhan Khalid, Husain Basrawala, knbk, Matt Seymour, Rod Xavier, scriptmonster, techydesigner, The_Cthulhu_Kid
47	Структура проекта	Antoine Pinsard, naveen.panwar, nicorellius
48	Теги шаблонов и фильтры	Antoine Pinsard, irakli khitarishvili, knbk, Medorator, naveen.panwar, The_Cthulhu_Kid
49	Тестирование устройства	Adrian17, Antoine Pinsard, e4c5, Kim, Matthew Schinckel, Maxime Lorant, Patrik Stenmark, SandroM, sudshekhar, Zags
50	Транзакции базы данных	Ian Clark
51	формы	Aidas Bendoraitis, Antoine Pinsard, Daniel Rucci, ettanany, George H., knbk, NBajanca, nicorellius, RamenChef, rumman0786, sudshekhar, trpt4him
52	Часовые пояса	William Reed
53	шаблонирование	Adam Starrh, Alasdair, Aniket, Antoine Pinsard, Brian H., coffee-grinder, doctorsherlock, fredley, George H., gkr, lxer, Stephen Leppik, Zags