ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНОГО ОКРУЖЕНИЯ

Для организации изолированного виртуального окружения python (virtual environment) можно использовать virtualenv (при помощи pip — package installer for Python).

Для создания виртуального окружения при помощи virtualenv необходимо, перейдя в директорию локального репозитория выполнить virtualenv *nameVE*. В результате будет создана директория с указанным именем и поддериректориями /bin, /lib и конфигурационным файлом pyvenv.cfg. Далее можно созданное изолированное окружение активировать (перейти в него), для этого необходимо выполнить `source nameVE/bin/activate`. В созданном изолированном окружении можно установить необходимое ПО, в частности Django требуемой версии (также используя pip).

Начиная с версии Python 3.4 вместе с ним идёт пакет venv, выполняющий те же функции — создание виртуальных окружений Python. Для создания используется соответствующая команда `python3 -m venv /path/to/new/virtual/environment`.

Такой подход может быть удобен для работы с несколькими проектами, требующими, например, различных версий тех или иных библиотек, для каждого из которых может быть создано своё виртуальное окружение.

DJANGO-ПРОЕКТ

Находясь в локальном репозитории необходимо создать Django-проект: `django-admin startproject projname`. В репозитории будет создана директория с соответствующим именем — *projname* (имя может быть любым, за исключением совпадающих с именами директорий Django проекта). Имя *projname* можно изменить при необходимости, оно является внешним и не является значимым для Django. Внутри будут созданы следующие файлы:

- manage.py: утилита командной строки для взаимодеqствия с Django (см. django-admin and manage.py).
- внутренняя директория с таким же именем *projname* представляет собой Python package для созданного проекта, это имя значимо и не должно изменяться.
- *projname*/__init__.py: инициализационный файл, указывающий, что это директория для Python package (см. more about packages).
- *projname*/settings.py: настроийки и конфигурация для Django проекта (см. Django settings)
- *projname*/urls.py: содержит объявления URL для Django проекта (“table of contents” of your Django-powered site, см. URL dispatcher).
- *projname*/asgi.py: кофигурация для использовния ASGI (см. How to deploy with ASGI).
- *projname*/wsgi.py: кофигурация для использовния WSGI (см. How to deploy with WSGI).

СТРУКТУРА DJANGO-ПРИЛОЖЕНИЯ

Django проект в общем случае представляет собой совокупность приложений и конфигурации сайта. В данном проекте может быть добавлено одно или несколько Web-приложений, представляющих собой некий определённый функционал. Таким образом, в одном проекте может быть несколько приложений, а одно приложение может использоваться в нескольких проектах. Для создания приложения необходимо выполнить команду: `python manage.py startapp appname`.

Django при создании приложения создаёт его базовую структуру директорий. Django-приложение использует паттерн MVT (Model-View-Template), в созданной директории содержатся соотвествующие файлы models.py и views.py.

Для корректной обработки запросов приложением необходимо создать конфигурационный файл URL, так называемый URLconf, с именем urls.py в директории проекта.

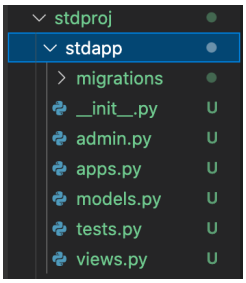
```python
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

Обработка запроса в Django-приложении происходит следующим образом (при поступлении запроса приложение определяет какой Python-код необходимо выполнить):

1. Django определяет используемый корневой модуль URLconf, как правило это значение



   параметра ROOT_URLCONF (параметр задаётся в файле settings.py, расположенном в директории проекта: ROOT_URLCONF = ‘stdproj.urls’, где stdproj.urls, указание на файл urls.py находящийся в директории Django-проекта). Корневой модуль URLconf может быть задан через атрибут urlconf входящего HttpRequest-объекта (устанавливается промежуточным программным обеспечением), и в этом случае будет использоваться значение атрибута вместо параметра ROOT_URLCONF.
2. Django загружает этот модуль Python и ищет переменную urlpatterns.
3. Django проходит по каждому шаблону URL по порядку и останавливается на первом соответствующем запрошенному URL-адресу шаблону (на основании значения атрибута path_info HttpRequest-объекта).
4. При совпадении шаблона URL Django импортирует и вызывает заданное View. Это может быть функция Python или class-based view. Django предоставляет подходящие для широкого круга приложений классы базовых представлений. Все представления наследуются от класса View, который позволяет обрабатывать связывание представления с URL-адресами, осуществлять отправку HTTP-запросов с соответствующими методами и т.д.
5. При возникновении ошибки на одном из шагов должна быть возвращена соответствующая ошибка.

ЗАПУСК DJANGO DEVELOPMENT SERVER

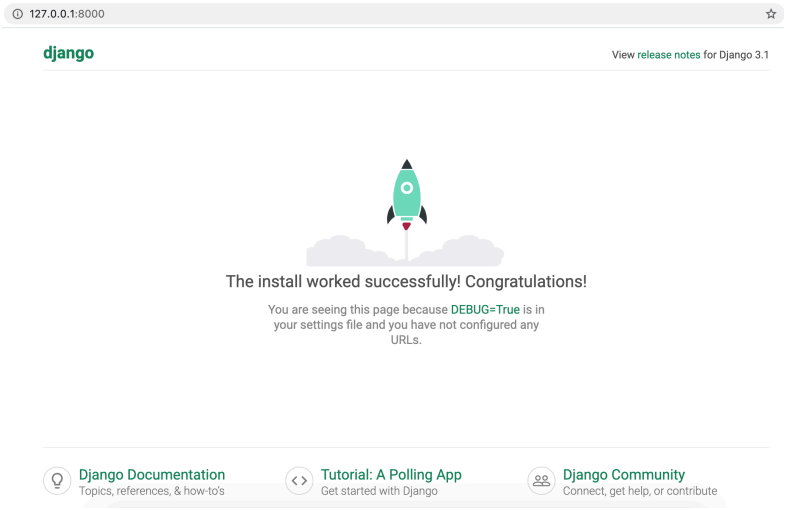
Для запуска Django development server необходимо перейти в директорию проекта: `cd projname` и выполнить команду `python manage.py runserver`. Команда должна выполняться именно из директории проекта, т.к. утилита manage.py находится там, вне её утилита недоступна.

Результат выполнения в консоли:

```
System check identified no issues (0 silenced).
April 02, 2021 - 04:33:40
Django version 3.1.7, using settings 'stdproj.settings'
Starting development server at http://127.0.0.1:8000/
```

По умолчанию команда runserver запускает сервер на порту 8000. Номер порта можно задать в качестве аргумента команды.

Открыв URI http://127.0.0.1:8000/ в браузере можно убелиться, что сервер запущен:



При этом при обращении к http://127.0.0.1:8000/ в консоли будут вводиться соответствующие логи:

```
[02/Apr/2021 04:34:43] "GET / HTTP/1.1" 200 16351
[02/Apr/2021 04:34:44] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Bold-webfont.woff HTTP/1.1" 200 86184
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Regular-webfont.woff HTTP/1.1" 200 85876
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Light-webfont.woff HTTP/1.1" 200 85692
```

Из логов видно, что при запуске сервера выполняются определённые HTTP-запросы с методом GET для получения требуемых данных.

ДОБАВЛЕНИЕ ПРИЛОЖЕНИЯ В ПРОЕКТ

Для добавления созданного приложения в проект необходимо указать его конфигурационный класс в переменной INSTALLED_APPS в файле настроек проекта settings.py:

1. В файле *appname*/apps.py декларируется конфигурационный класс приложения с именем

```
class StdappConfig(AppConfig):
    name = 'stdapp'
```

2. В файле проекта settings.py в переменной INSTALLED_APPS указываются все приложения проекта, при добавлении нового приложения, необходимо добавить в данную переменную ссылку на его конфигурационный класс, путь указывается через точку: *директория приложения.содержащий класс файл.название класса*.

```
INSTALLED_APPS = [
    'stdapp.apps.StdappConfig',
    'django.contrib.admin',
    'django.contrib.auth',
```

БАЗА ДАННЫХ

По умолчанию Django использует SQLite, она включена в проект и не требует установки.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

При необходимости использовать другую СУБД необходимо в файле настроек проекта settings.py в DATABASES изменить значение на требуемое и задать дополнительные параметры: USER, PASSWORD, HOST, PORT (см. DATABASES).

Django поддерживает:

- PostgreSQL (ENGINE : ’django.db.backends.postgresql'),
- MySQL (jango.db.backends.mysql),
- Oracle (django.db.backends.oracle).

Также может быть использован ряд других СУБД, в частности Microsoft SQL Server некоторых версий, но для них ряд предоставляемых ORM-функций может достаточно сильно отличаться от тех, что предоставляются официально поддерживаемым СУБД.

MODEL И MIGRATIONS

Модель в Django представляет собой источник информации о данных. Она содержит основные поля и поведение хранимых данных. Как правило, каждая модель отображается в одну таблицу базы данных. В Django модели представлены классами Python, т.е. конкретная сущность описывается отдельным классом. При этом каждая модель представляет собой подклассы Python django.db.models.Model, соответственно, могут быть использованы методы Model. Атрибут модели — поле таблицы в базе данных. Для работы с моделями Django предоставляет автоматически генерируемый API доступа к базе данных (Model API reference).

```
class Note(models.Model):
    note_text = models.CharField(max_length=200)
```

Поля модели представлены экземплярами класса Field, он определяет тип поля, так например для символьных полей используют CharField, для десятичных чисел DecimalField и т.д.

После того как модель описана, её необходимо активировать, для это необходимо выполнить команду `python manage.py makemigrations` *`appname`*. Т.е. выполнить миграцию, в Django миграция описывает, каким образом Django сохраняет изменения в моделях и, соотвественно, в схеме базы данных.

При успешном выполнении команды в консоль будет выведено:

```
Migrations for 'stdapp':
  stdapp/migrations/0001_initial.py
    - Create model Note
```

где stdapp — название приложения, а Note — имя модели. Все описанные модели в файле приложения models.py будут активированы и также перечислены в выводе выполнения команды `python manage.py makemigrations`.

Выполненную миграцию можно посмотреть в сгенерированном файле *appname*/ migrations/0001_initial.py, где 0001 — номер выполненной миграции:

```python
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Note',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True,
                ('note_text', models.CharField(max_length=200)),
            ],
        ),
    ]
```

Данный файл при необходимости может быть отредактирован вручную.

Помимо описанных в модели полей дополнительно будет создано поле id. При добавлении новых описаний моделей или изменений в имеющихся, миграции будут выполняться только для внесённых изменений.

Для автоматического запуска миграций используется команда `migrate`, команда имеет два параметра [app_label] и [migration_name], по умолчанию будут выполнены все миграции для всех приложений, при указании параметра [app_label] для конкретного приложения (также может запустить связанные миграции в других приложениях), при указании [migration_name], будет выполнена конкретная миграция, при этом более поздние миграции выполнены не будут (# см. Migrations).

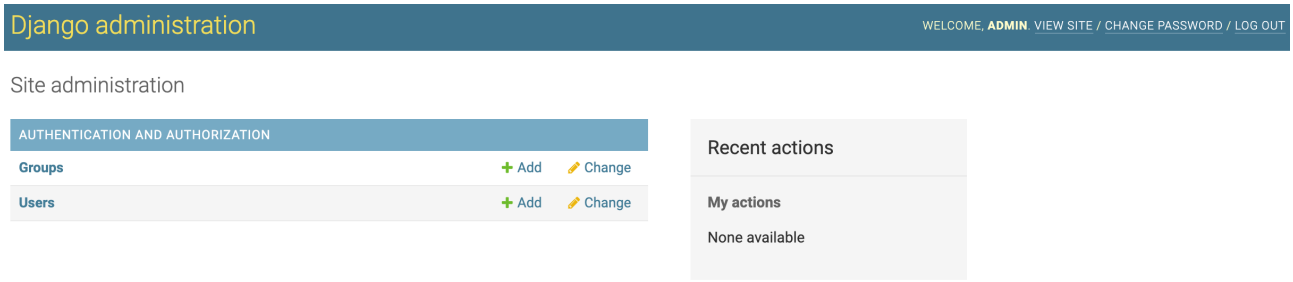
Для просмотра SQL запроса миграции можно воспользоваться командой `python manage.py sqlmigrate` *`appname`* `0001` (где 0001 — номер миграции):

```
BEGIN;
--
-- Create model Note
--
CREATE TABLE "stdapp_note" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "note_text" varchar(200) NOT NULL);
COMMIT;
```

ПОЛЬЗОВАТЕЛЬ-АДМИНИСТРАТОР

Для создания пользователя администратора используется команда `python manage.py createsuperuser`. После будет предложено ввести желаемое имя, по умолчанию это будет имя пользователя в системе (если оставить строку ввода пустой), указать эл. почту и задать пароль. После того как пользователь-администратор создан можно запустить сервер (`python manage.py runserver`) и в браузере перейти http://127.0.0.1:8000/admin/ .

В браузере будет загружен соотвествующий ресурс:



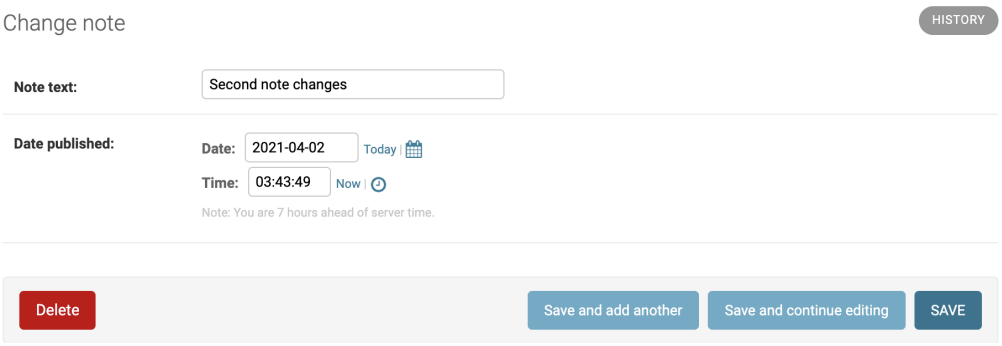
Для добавления интерфейса администратора для приложения необходимо в файле приложения admin.py указать модели приложения, которые должны быть доступны администратору — admin.site.register(*appname*):

```
from django.contrib import admin

from .models import Note

admin.site.register(Note)
```

Через интерфейс администратора можно добавлять новые объекты, редактировать и удалять уже имеющиеся.



А также просматривать историю изменений (“History”).

ИСПОЛЬЗОВАНИЕ TEMPLATE ДЛЯ ПРЕДСТАВЛЕНИЯ ДАННЫХ МОДЕЛИ

Django использует MVT подход и для представления данных моделей используется шаблоны, их основное предназначение — динамическое представление данных. Шаблоны представляют собой статический HTML (либо в другом поддерживаемом формате XML, CSV

и т.д.) и динамические данные, рендеринг которых описывается специальным синтаксисом. Для размещения шаблонов в директории приложения необходимо создать поддиректорию template, где должны быть размещены файлы, содержащие шаблоны.

Произвольную директорию можно указать в соответствующей переменной TEMPLATE — DIR[ ]. При указании 'APP_DIRS': True, загрузчик шаблонов будет искать шаблоны в каталоге шаблонов приложения.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
```

Основными конструкциями шаблонов являются теги и переменные.

Переменная позволяет выводить некое значение контекста, которое, по сути, является dict-подобным объектом (изменяемый объект-отображение): {{ *variable name* }}. Контекст в данном случае — словарь содержащий пары ключ-значение, где имена переменных и их значения выступают, соответственно, в качестве ключа и его значения. Для обращения и поиска используется точечная нотация: *some_object.attribute*.

Теги обеспечивают произвольную логику в процессе рендеринг. Тег задаётся следующей конструкцией: {% *name* %}. Теги могут использоваться для вывода контекста, представлять структуру управления, например для организации цикла. Часть тегов требует открывающей и закрывающей конструкции, так парный тег используется для организации таких циклов как if и for:

{% if … %} {% endif %} {% else %}
{% for … %} {% endfor %}

К шаблону могут применяться фильтры: {{ some_date|date:"Y-m-d" }}, где date — фильтр, а “Y-m-d" — аргумент фильтра (но, большая часть фильтров Django не принимает аргументы).

Для комментариев имеется соотвествующий тег:
{# comment #} — для однострочных комментариев;
{% comment %} — для многострочных комментариев.

Django содержит достаточно большое число различных тегов и фильтров , и также позволяет создавать пользовательские теги и пользовательские фильтры. Стоит обратить внимание, что в языке шаблонов Django нет обработки исключений, при получении исключения выдаётся ошибка сервера.

Для представления данных пользователю посредством шаблонов, необходимо обратиться к шаблону во View (в представленном примере шаблон “Notes.html”):

```
from django.shortcuts import render
from django.http import HttpResponse
from django.template import loader

from .models import Note

def index(request):

    notes_list = Note.objects.all()
    template = loader.get_template('Notes.html')
    context = {
        'notes_list': notes_list,
    }
    return HttpResponse(template.render(context, request))
```

Данный View будет представлять пользователю список сохранённых заметок из Note. Здесь запрашиваются все имеющиеся объекты. Для вывода текста заметок используется шаблон:

```
{% if notes_list %}
    {% for note in notes_list %}
        <ul class="list-group">
            <li class="list-group-item">
                {{ note.note_text }}
            </li>
    {% endfor %}
        </ul>
{% else %}
    <p>No one note are available.</p>
{% endif %}
```

где {{ note.note_text }} — переменная, чьё значение будет выведено пользователю, в данном случае это будет значение атрибута note_text объекта note.

ОБНОВЛЕНИЕ ДАННЫХ МОДЕЛИ

Для отображения данных пользователю приложения используются теплейты (templates), посредством которых представление отображает данные модели. Сама модель представляет собой соответствующую таблицу в БД (базе данных), а конкретная запись в таблице является экземпляром модели. Данные модели могут обновляться различными способами, в частности через интерфейс администратора или непосредственно пользователем. В Django для получения данных для изменений в БД, в общем, случае используются формы. Они позволяют создавать, обновлять и удалять экземпляры модели.

HTML-формы

HTML-формы представляют собой набор элементов помещённых в тег <form>…</form>, в частности включают в себя виджеты для ввода различных типов данных (текст, дата, ссылка и т.д.). Для работы с элементами HTML-формы как и для HTML-документа в целом применяются CSS и JavaScript. Как правило, форма содержит <input> элементы, в том числе <input type="submit">, элемент типа "submit" используется для отправки данных формы в соотвествующий обработчик формы.

```
<form method="POST">
    <label for="team_name">Note: </label>
    <input id="notename" type="text" name="name_field" value="Note header">
    <label for="team_name">Note text: </label>
    <input id="notetext" type="text" name="name_field" value="Text">
    <input type="submit" value="OK">
</form>
```

Тег <form> имеет два важных атрибута method и action. Атрибут метод определяет какой HTTP-метод требуется использовать, для форм используются методы POST или GET:

- метод POST используется в случае, если требуется отправить данные для внесения изменений в БД;
- метод GET используется в случае, если требуется получить данные, в том числе для выполнения запросов к БД.

Атрибут action определяет ресурс/URL-адрес куда требуется отправить данные для обработки. В случае если атрибута не задано, введённые данные будут отправлены в код представления (функцию, или класс), сформировавший текущую страницу.

В общем случае, на сервер возлагается отправка начального состояния формы, обработка полученных от клиента данных, в частности их валидация. В случае, если данные не корректны сообщить об этом пользователь, например, отправив вновь начальное состояние формы и соответствующее сообщение с описанием проблемы. В случае, если данные не корректны сервер должен выполнить предусмотренные действия, например, сохранить данные, возвратить результата поиска, загрузить файл и т.д. И при необходимости проинформировать пользователя о совершённых действиях.

Формы в Django

Использование HTML-форм относится в большей степени к темплейтам (templates), в Представлениях (views) в Django используются собственные формы, определяемые через класс Form, который позволяет создавать HTML-формы: описывает форму и определяет, как форма работает и выглядит.

Как правило, в директории проекта создаётся соотвествующий файл forms.py:

```
forms.py
from django import forms

class UserForm(forms.Form):
    username = forms.CharField(max_length=50)
    password = forms.CharField(max_length=20)
```

Каждое поле формы определенно своим классом, например, FileField, DateField, TextField и т.д. (см. Field types) и представлено соответствующим виджетом. Так, например, поле обозначенное как CharField по умолчанию представлено виджетом TextInput, который создает тег <input type="text"> в HTML. Если необходимо использовать другой виджет он может быть предопределён при определении поля формы: вместо forms.CharField() можно указать forms.CharField(widget=forms.Textarea), при этом будет создан тег <textarea> вместо тега <input type="text">.

```
forms.py
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    comment = forms.CharField(widget=forms.Textarea)
```

Форма рендерится для клиента сходным с рендерингом данных модели образом:

- в Представлении определён соотвествующий код для формы;

```
userform = UserForm()
return render(request, "index2.html", {"form": userform})
```

- в используемом темплейте указана соответствующая переменная контекста ({{ form }}, указанный в данном примере {% csrf_token %} используется для защиты от межсайтовой подделки запроса, не рекомендован для использования с ссылающимися на внешние URL формами); при этом все поля формы будут добавлены в HTML из переменной контекста {{ form }} при рендеринге шаблона.

```
<form method="POST">
    {% csrf_token %}
<table>
    {{ form }}
</table>
```

Если для <form> не указан атрибут action, то данные формы будут возвращены в тот же фрагмент кода, из которого была вызвана форма, если атрибут определён, то по указанному в атрибуте URL.

Для рендеринга полей формы в тегах <tr>, <p>, <li> могут быть заданы соотвествующие значения переменной контекста:

- {{ form.as_table }} — поля формы будут выведены в таблице, в ячейках тега <tr>, при этом тег <table> в HTML должен быть создан заранее;
- {{ form.as_p }} — поля формы будут выведены в теге <p>
- {{ form.as_ul }} — поля формы будут выведены в теге <li>, при этом тег <ul> в HTML должен быть создан заранее.

При необходимости конкретные поле можно рендерить вручную, доступ к полю можно получить через атрибут формы name_of_field: {{ form.name_of_field }}.

При использовании метода GET Представление должно добавлять в контекст шаблона пустую форму для рендеринга и отправки. При запросе POST представление можно отправлять форму с соответствующими данными из запроса, при этом форма будет связана с этими данными. Для уточнения связана ли форма с данными можно проверить значение атрибута is_bound.

Класс Form имеет метод для проверки данных формы is_valid() для проверки валидности данных формы. Он возвращает True/False и в зависимости от результата пользователю должна быть возвращена форма с требованием ввести корректные данные (возвращено False) или же при необходимости сообщение об успешном завершении действия (возвращено True). При этом так как данные и формы связаны, то можно вернуть пользователю форму с введёнными ранее данными с указанием какие из них некорректны. Помимо метода is_valid() для проверки валидности данных существует множество различных методов (см. Form and field validation). Для работы с ошибками формы существуют соотвествующие инструменты (Form.errors).

Ошибки форм также требуют соответствующего рендеринга, например, можно использовать {{ form.non_field_errors }} для работы с ошибками, не относящихся к определённому полю, так же можно получить ошибку по конкретному полю через атрибут errror ({{ form.fieldname.errors }}).

```
{% if form.non_field_errors %}
<ul>
  {% for error in form.non_field_errors %}
    <li>{{ error }}</li>
  {% endfor %}
</ul>
{% endif %}
```

Создание формы на основе модели

Как правило, приложение использует базу данных, и многие формы аналогичны моделям. И, соответственно, создание формы дублирующую модель избыточно, поэтому в Django имеется возможность создавать формы на основе классов модели. Для этого используется класс ModelForm.

```
from django.forms import ModelForm
from myapp.models import Note

class NoteForm(ModelForm):
    class Meta:
        model = Note
        fields = ['note_name','note_text','pub_date']
```

При создании формы на основании модели необходимо указать модель и перечислить соотвествующие поля модели. Сгенерированный класс формы будет содержать соответствующее поле формы для каждого поля модели в том порядке, в котором они указаны в атрибуте fields. При этом каждому полю модели соответствует стандартное поле формы. Так, поле модели CharField будет представлено на форме как CharField. Но не все классы полей совпадают в моделях и формах, так поле модели ManyToManyField будет представлено как поле формы MultipleChoiceField. Соответствие между квасами полей можно посмотреть в Field types.

Для созданной на основе модели формы также проводится валидация. Для этого также может быть использован is_valid() для выполнения проверки всех полей, включенных в форму. Для проверки полей модели используется Model.clean_fields(), для проверки объекта модели целиком — Model.clean(), проверка уникальности полей проводится при помощи Model.validate_unique().

МЕНЕДЖЕР МОДЕЛИ DJANGO

Для работы с базой и обеспечения операций с данными в базе модель использует соответсвующий интерфейс — менеджер модели. По умолчанию Django для каждого класса модели добавляет Manager с именем objects. Имя менеджера можно изменить, для этого необходимо определить для требуемого класса атрибут класса типа models.Manager() с требуемым именем (model_manager_name = models.Manager()). Используя менеджер модели, можно обращаться к конкретным полям объектов класса модели.

ОПЕРАЦИИ С ОБЪЕКТАМИ И ДАННЫМИ МОДЕЛИ

Модель неразрывно связана с базой данных и, следовательно, работа с моделью подразумевает выполнение запросов к базе данных. Django предоставляет API-интерфейс для базы данных, который позволяет создавать, извлекать, обновлять и удалять объекты.

Для получения объектов из базы данных необходимо создать QuerySet, используя менеджер соответствующего класса модели. QuerySet представляет коллекцию объектов из базы данных. Например, Note.objects.all() позволяет получить все объекты класса Note, хранящиеся в базе данных. Для работы с QuerySet используется соотвествующий API.

Для получения конкретных объектов или конкретных групп объектов используются фильтры по заданным параметрам filter(**kwargs), он возвращает новый QuerySet, содержащий объекты, которые соответствуют заданным параметрам поиска, например, Note.objects.filter(pub_date__year=2020) вернёт все записи за 2020 год. QuerySet можно ограничить при помощи соответствующего синтаксиса нарезки массивов в Python, например, Note.objects.all()[:5], вернёт массив из первых пяти объектов. Или же можно извлечь конкретный объект Note.objects.all()[0], при необходимости объекты можно упорядочить и выбрать конкретный из них, например, Entry.objects.order_by('note_name')[0] и т.д.

Также может быть использован метод exclude(**kwargs), он возвращает новый QuerySet, содержащий объекты, которые не соответствуют указанным параметрам поиска. Данные методы могут использоваться в комбинации при необходимости.

Если соотнести QuerySet с SQL, то его получение приравнивается к работе оператора SELECT, а фильтр при этом выступает в качестве ограничивающего предложения, как, например, WHERE (# …(pub_date__year=2020)) или LIMIT (# …[:5]).

Для работы с объектами модели в Django используются различные методы, для простых действий таких как создание обновление и удаления объекта применяются:
save() — для создания и сохранения изменений объектов модели;
delete() — соотвественно, для удаления объектов.

Для работы с объектами в наборах связанных объектов (объектов, находящихся в отношениях "один ко многим" или "многие ко многим"):

add() — для добавления объектов в набор связанных объектов используется метод например, при сохранении ForeignKey и ManyToManyField;
create() — для создания объекта помещаемого в связаный список объетов;
remove() — соотвественно, для его удаления.

Помимо данных методов для работы с моделью имеется множество различных методов, они описаны в соответствующей документации (в частности, см. Models, QuerySet).