**Тема занятия № 30: Модуль 16. Модели: расширенные инструменты**

1. Полиморфные связи

Полиморфная, или обобщенная, связь позволяет связать запись вторичной модели, в которой она объявлена, с записью любой модели, имеющейся в приложениях проекта, без исключений. При этом разные записи такой модели могут оказаться связанными с записями разных моделей.

Предположим, что мы хотим дать посетителю возможность оставлять заметки к рубрикам, объявлениям, машинам и составным частям. Вместо того, чтобы создавать четыре совершенно одинаковые модели rubricnote, bbnote, machinenote И Sparenote и связывать их с соответствующими моделями обычными связями "один со-многими”, мы можем написать всего одну модель Note и объявить в ней полиморфную связь.

Перед созданием полиморфных связей нужно убедиться, что приложение Dj ango. Contrib. Contenttypes, реализующее функциональность соответствующей подсистемы Django, присутствует в списке зарегистрированных приложений (параметр installed apps настроек проекта). После этого следует хотя бы раз провести выполнение миграций, чтобы Django создал в базе данных необходимые таблицы.

На заметку!

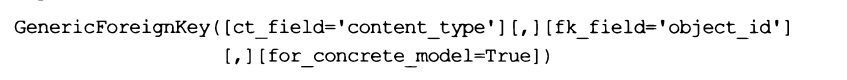
Для хранения списка созданных в проекте моделей, который используется подсистемой полиморфных связей в работе, в базе данных формируется таблица django Content type. Править ее вручную не рекомендуется.

Полиморфная связь создается в классе вторичной модели. Для ее установления необходимо объявить там три сущности:

□ поле для хранения типа модели, связываемой с записью. Оно должно иметь тип Foreignkey (т. Е. Внешний ключ для связи "один-со-многими"), устанавливать СВЯЗЬ С моделью contenttype ИЗ модуля django. Contrib. Contenttypes.models (там хранится перечень всех моделей проекта) и выполнять каскадное удаление. Обычно такому полю дается имя content type;

□ поле для хранения значения ключа связываемой записи. Оно должно иметь целочисленный тип — обычно positiveintegerfieid. Как правило, этому полю дается имя object id;

□ поле полиморфной Связи, реализуемое экземпляром класса genericforeignkey из модуля django.contrib.contenttypes, fields. Вот формат вызова его конструктора:



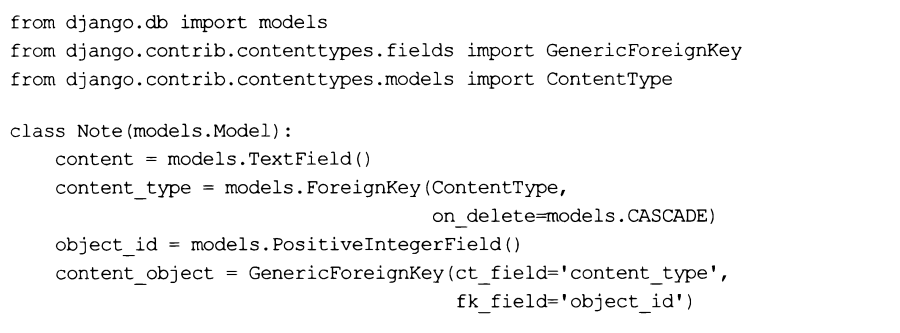
Конструктор принимает следующие параметры:

• ct fieid— указывает имя поля, хранящего тип связываемой модели, если это имя отличается от content type;

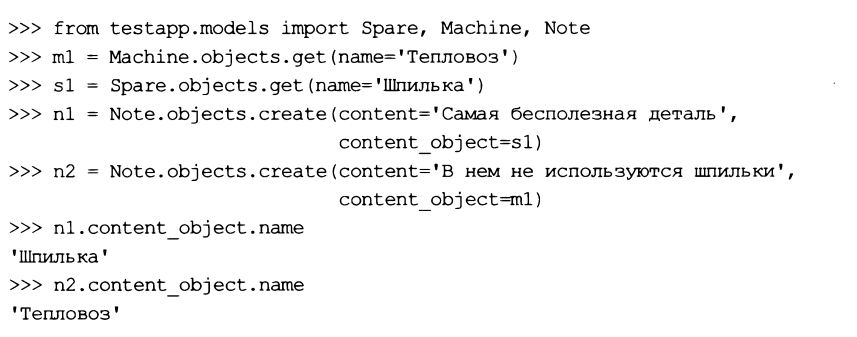
• fk\_fieid — указывает имя поля, хранящего ключ связываемой записи, если это имя отличается от object id;

• for concrete modei— следует дать значение False, если необходимо устанавливать связи, в том числе и с прокси-моделями (о них будет рассказано позже).

Cодержит код модели Note, хранящей заметки и использующей для связи с соответствующими моделями полиморфную связь.

****

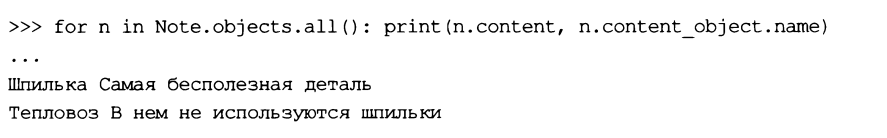
Чтобы связать запись модели, содержащей полиморфную связь, с записью другой модели, последнюю следует присвоить полю полиморфной связи. Для примера создадим две заметки — для шпильки и тепловоза:

****

Из поля, хранящего тип связанной модели, можно получить объект записи, описывающей этот тип. А обратившись к полю name этой записи, мы получим сам тип связанной модели, фактически — имя ее класса, приведенное к нижнему регистру:

****

Переберем в цикле все заметки и выведем их текст вместе с названиями связанных сущностей:

****

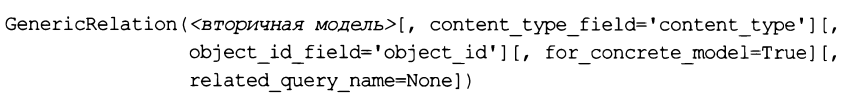
К сожалению, поле полиморфной связи нельзя использовать в условиях фильтрации. Так что вот такой код вызовет ошибку:



При создании полиморфной связи Django по умолчанию не предоставляет средств для доступа из первичной модели к связанным записям вторичной модели. Такие средства придется создавать самостоятельно.

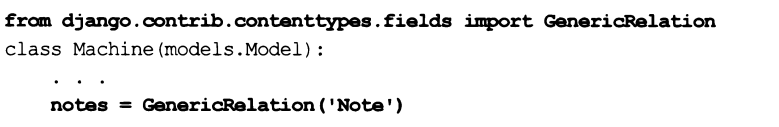
Доступ из записи первичной модели к связанным записям вторичной модели в случае полиморфной связи предоставляет так называемое поле обратной связи. Оно реализуется экземпляром класса genericrelation ИЗ модуля django. Contrib.

Contenttypes.fields. Конструктор этого класса вызывается в формате:

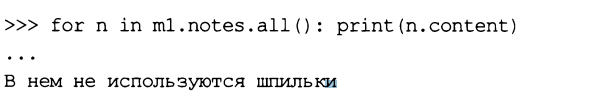
****

Вторичная модель указывается либо в виде ссылки на класс, либо в виде строки с именем класса. Параметр content type field указывает имя поля, хранящего тип связываемой модели, а параметр object id fieid — имя поля с ключом связываемой записи, если эти имена отличаются от используемых по умолчанию Content type и object id соответственно. Если одна из связываемых моделей является прокси-моделью, то параметру for\_concrete\_model нужно присвоить значение False. Назначение параметра related\_query\_name аналогично таковому у поля типа Foreignkey.

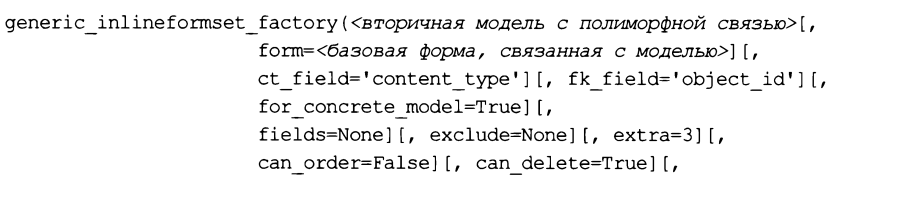
Чтобы из записи модели Machine получить список связанных заметок, нам нужно добавить в объявление ее класса такой код:

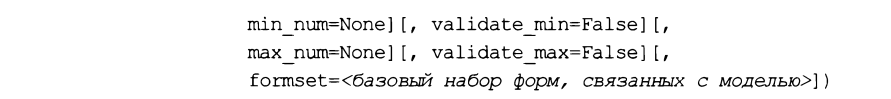
****

Поле обратной связи хранит набор связанных записей вторичной модели. Например, так мы можем перебрать все заметки, оставленные для тепловоза, и вывести на экран их содержимое:

****

Поля типа genericforeingkey никак не представляются в формах, связанных с моделями. Однако Django позволяет создать встроенный набор форм, позволяющий работать со связанными записями (подробнее о встроенных наборах форм рассказывалось ранее). Он создается посредством функции generic\_Inlineformset\_factory () ИЗ МОДУЛЯ django. Contrib. Contenttypes. Forms, CO следующим форматом вызова:





Параметр ct fieid указывает имя поля, хранящего тип связываемой модели, а параметр fk\_fieid — имя поля, в котором сохраняется ключ связываемой записи, если ЭТИ имена отличаются ОТ используемых ПО умолчанию content type И object id соответственно. Если одна из связываемых моделей является прокси-моделью, параметру for\_concrete\_modei нужно дать значение False. Базовый набор записей, указываемый в параметре formset, должен быть производным от класса Basegenericinlinefoxmset ИЗ модуля django. Contrib. Contenttypes. Forms.

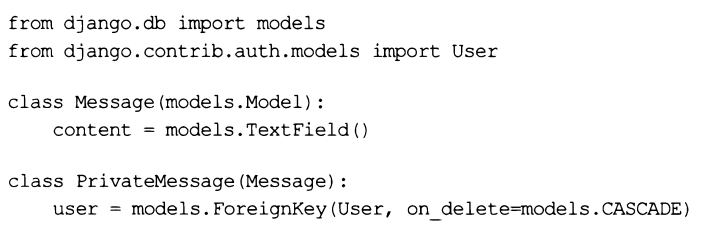
1. Наследование моделей

Модель Django — это обычный класс Python. Следовательно, мы можем объявить модель, являющуюся подклассом другой модели. Причем фреймворк предлагает нам целых три способа сделать это.

**Прямое наследование моделей**

В случае прямого, или многотабличного, один класс модели просто наследуется от другого.

Код двух моделей: базовой Message, хранящей сообщения, и производной privatemessage, которая хранит частные сообщения для зарегистрированных пользователей.



В этом случае Django поступит следующим образом:

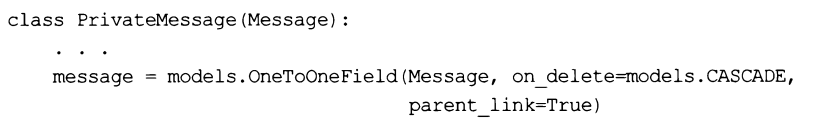
□ создаст в базе данных две таблицы — для базовой и производной моделей. Каждая из таблиц будет включать только те поля, которые объявлены в соответствующей модели;

□ установит между моделями связь "один-с-одним". Базовая модель станет первичной, а производная — вторичной;

О создаст в производной модели поле с именем вида <имя базовой модели>^р^г.

Реализующее связь "один-с-одним" с базовой моделью.

Такое служебное поле можно создать вручную, дав ему произвольное имя, указав для него каскадное удаление и обязательно присвоив его параметру Parent link значение True. Пример:



□ создаст в каждом объекте записи базовой модели атрибут с именем вида <имя производной модели>. Хранящий объект записи производной модели;

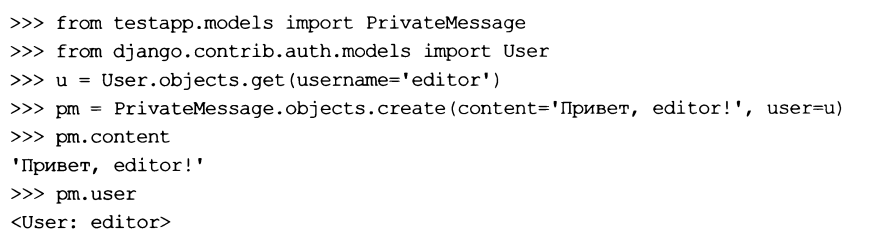
□ при сохранении данных в записи производной модели — сохранит значения полей, унаследованных от базовой модели, в таблице базовой модели, а значения полей производной модели — в таблице производной модели;

□ при удалении записи — фактически удалит обе записи: из базовой и производной моделей.

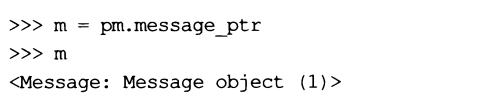
Есть возможность удалить запись производной модели, оставив связанную запись базовой модели. Для этого при вызове у записи производной модели метода delete () следует указать в нем параметр keep parents со значением True.

Производная модель наследует от базовой все параметры, заданные во вложенном классе Meta. При необходимости эти параметры можно переопределить в производной модели.

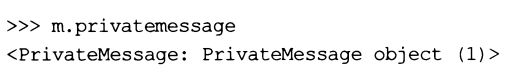
Для примера добавим в модель privatemessage запись и получим значения ее полей:

****

Получим связанную запись базовой модели Message:

****

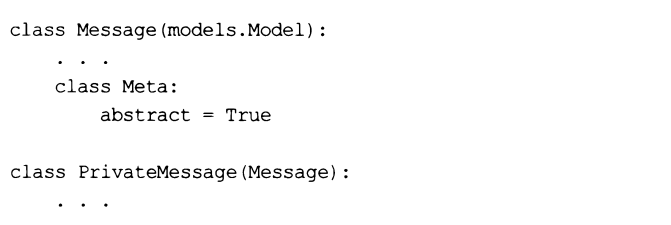
Получим запись производной модели из записи базовой модели:

****

Прямое наследование может выручить, если нужно хранить в базе данных набор сущностей с примерно одинаковым набором полей. Тогда поля, общие для всех типов сущностей, выносятся в базовую модель, а в производных объявляются только поля, уникальные для каждого конкретного типа сущностей.

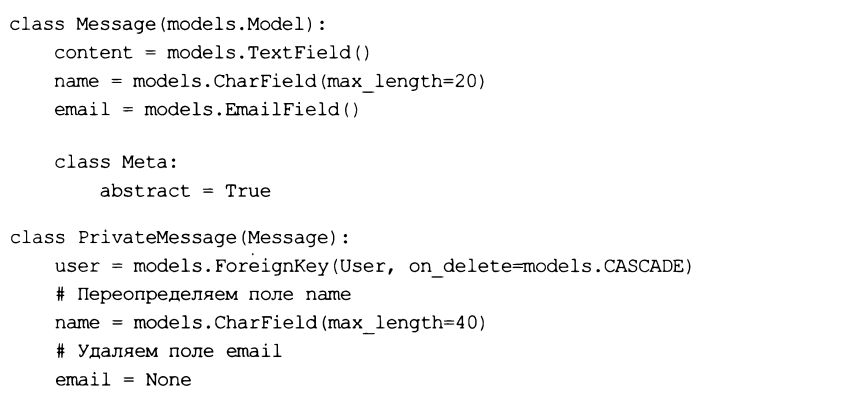
**Абстрактные модели**

Второй способ наследовать одну модель от другой — пометить базовую модель как абстрактную, задав в ней (т. Е. Во вложенном классе Meta) параметр abstract со значением True. Пример:

****

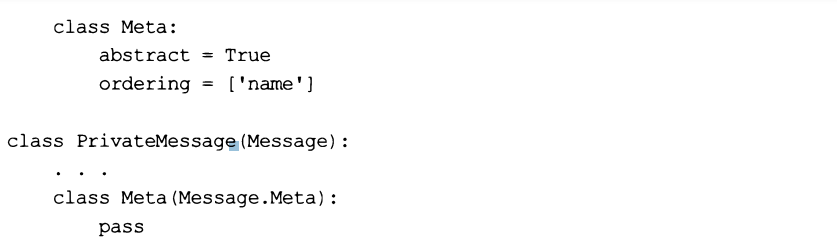
Для абстрактной базовой модели в базе данных не создается никаких таблиц. Напротив, таблица, созданная для производной от нее модели, будет содержать весь набор полей, объявленных как в базовой, так и в производной модели.

Поля, объявленные в базовой абстрактной модели, могут быть переопределены в производной. Можно даже удалить объявленное в базовой модели поле, объявив в производной модели атрибут класса с тем же именем и присвоив ему значение None. Пример такого переопределения и удаления полей приведен.

****

Параметры абстрактной базовой модели, объявленные во вложенном классе Meta, не наследуются производной моделью автоматически. Однако есть возможность унаследовать их, объявив вложенный класс Meta как производный от класса Meta базовой модели. Пример:





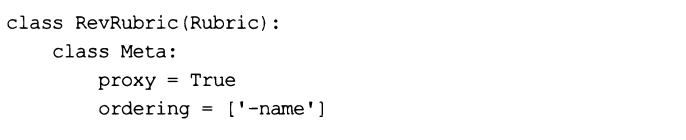
Наследование с применением абстрактной базовой модели применяется в тех же случаях, что и прямое наследование. Оно предлагает более гибкие возможности по объявлению набора полей в производных моделях— так, мы можем переопределить или даже удалить какое-либо поле, объявленное в базовой модели. К тому же, поскольку все данные хранятся в одной таблице, работа с ними выполняется быстрее, чем в случае прямого наследования.

**Прокси-модели**

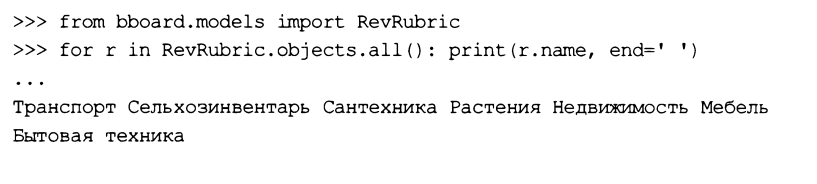
Третий способ — объявить производную модель как прокси-модель. Классы такого типа не предназначены для расширения или изменения набора полей базовой модели, а служат для расширения или изменения ее функциональности.

Прокси-модель объявляется заданием в параметрах производной модели (во вложенном классе Meta) параметра proxy со значением True. В базовой модели при этом никаких дополнительных параметров записывать не нужно.

Для прокси-модели не создается никаких таблиц в базе данных. Все данные хранятся в таблице базовой модели. КОД прокси-модели revrubric, производной от модели Rubric и задающей сортировку рубрик по убыванию названия.

****

Для примера выведем список рубрик из только что созданной прокси-модели:

****