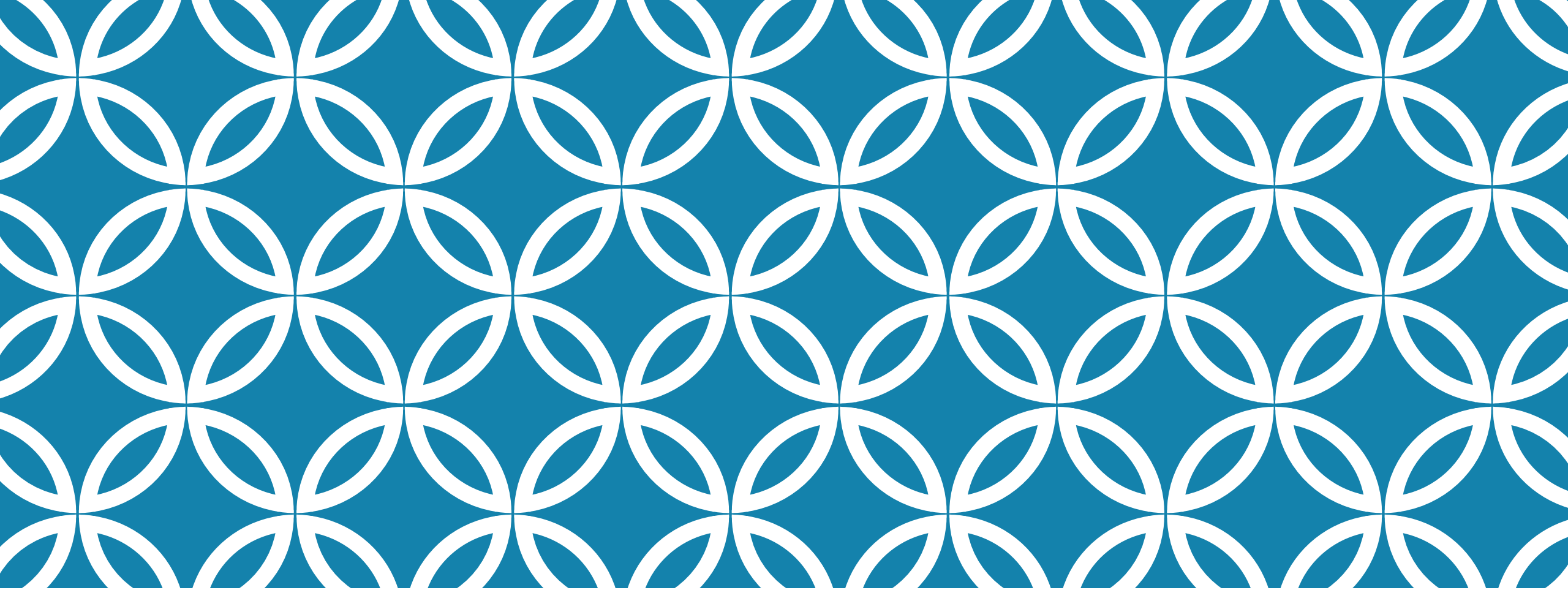


СОЗДАНИЕ ПЕРВОГО ПРОЕКТА

САЙТ АДМИНИСТРИРОВАНИЯ

- Следующей ниже командой запустите сервер разработки

```
python manage.py runserver
```

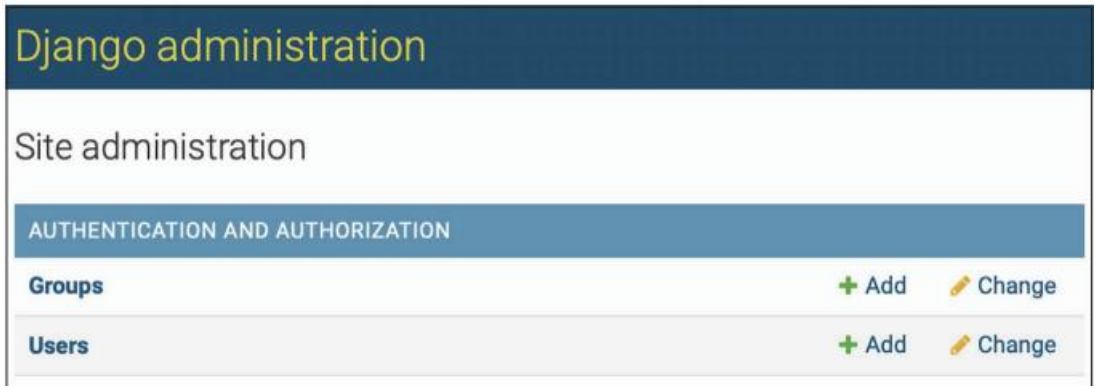
Пройдите по URL-адресу http://127.0.0.1:8000/admin/ в своем браузере. Вы должны увидеть страницу входа на сайт администрирования, как показано на рис.

Экран входа на сайт администрирования

САЙТ АДМИНИСТРИРОВАНИЯ

Войдите на сайт администрирования, используя учетные данные пользователя, которые вы создали на предыдущем шаге. Вы увидите индексную страницу сайта администрирования, как показано на рисунке.

Модели **Group** и **User**, которые вы видите на приведенном выше скриншоте, являются частью встроенного в Django фреймворка аутентификации, расположенного в **django.contrib.auth**. Если кликнуть по **Users** (Пользователи), то можно увидеть пользователя, которого вы создали ранее



Индексная страница сайта администрирования

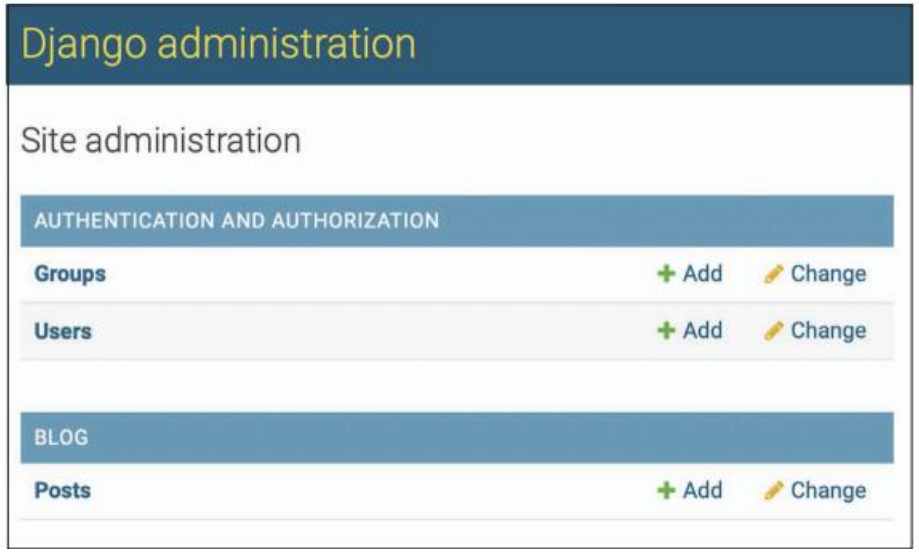
ДОБАВЛЕНИЕ МОДЕЛЕЙ НА САЙТ АДМИНИСТРИРОВАНИЯ

Добавляем модель блога на сайт администрирования. Отредактируйте файл admin.py приложения blog, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Перезагрузить сайт администрирования в своем браузере. Вы должны увидеть свою модель Post на сайте, как показано ниже:

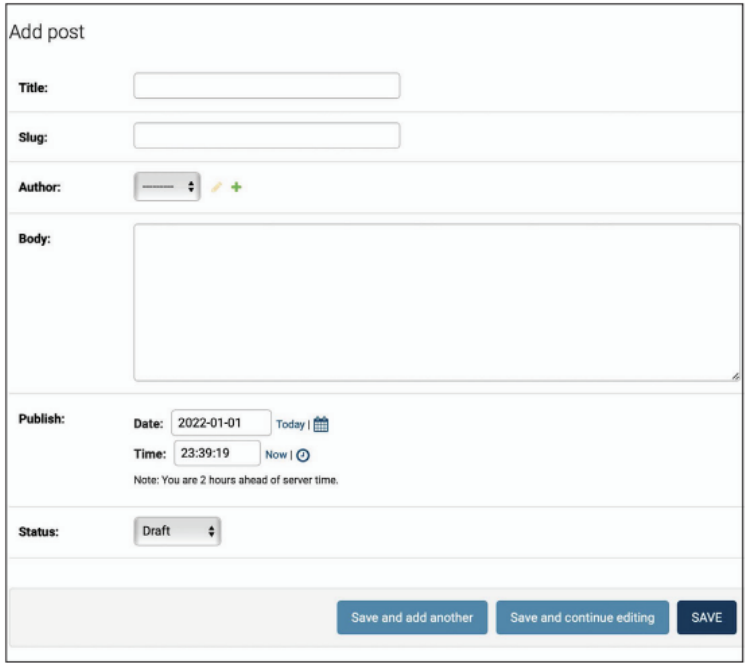


Модель Post приложения blog

ДОБАВЛЕНИЕ МОДЕЛЕЙ НА САЙТ АДМИНИСТРИРОВАНИЯ

При регистрации модели на сайте администрирования будет получен удобный интерфейс, сгенерированный путем интроспекции созданных разработчиком моделей, позволяющий простым способом выводить списки, редактировать, создавать и удалять объекты.

Кликните по ссылке **Add** (Добавить) напротив **Posts** (Посты), чтобы добавить новый пост. Вы увидите форму, которую Django сгенерировал для модели динамически, как показано на рис.



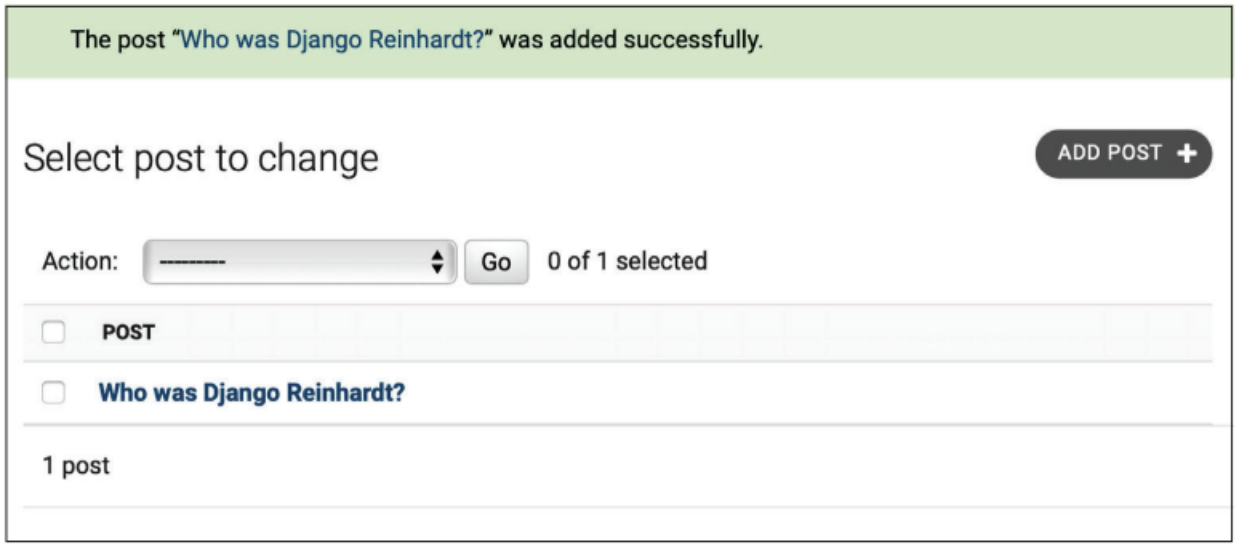
Форма редактирования на сайте администрирования для модели Post

ДОБАВЛЕНИЕ МОДЕЛЕЙ НА САЙТ АДМИНИСТРИРОВАНИЯ

Для каждого типа поля Django использует различные виджеты форм.

Даже сложные поля, такие как поле **DateTimeField**, отображаются на странице с простым интерфейсом, таким как элемент выбора даты на языке JavaScript.

Заполните форму и кликните по кнопке **Save** (Сохранить). Вы будете перенаправлены на страницу списка постов с сообщением об успехе и только что созданным постом, как показано на рис.



Представление списка на сайте администрирования для модели Post с сообщением об успешном добавлении

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Давайте посмотрим на способы адаптации сайта администрирования под конкретно-прикладную задачу. Отредактируйте файл **admin.py** приложения **blog**, изменив его, как показано ниже. Новые строки выделены жирным шрифтом.

```
from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
```

Мы сообщаем сайту администрирования, что модель зарегистрирована на сайте с использованием конкретно-прикладного класса, который наследует от **ModelAdmin**.
В этот класс можно вставлять информацию о том, как показывать модель на сайте и как с ней взаимодействовать.
Атрибут **list_display** позволяет задавать поля модели, которые вы хотите показывать на странице списка объектов администрирования.

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Декоратор **@admin.register()** выполняет ту же функцию, что и функция **admin.site.register()**, которую вы заменили, регистрируя декорируемый им класс **ModelAdmin**.

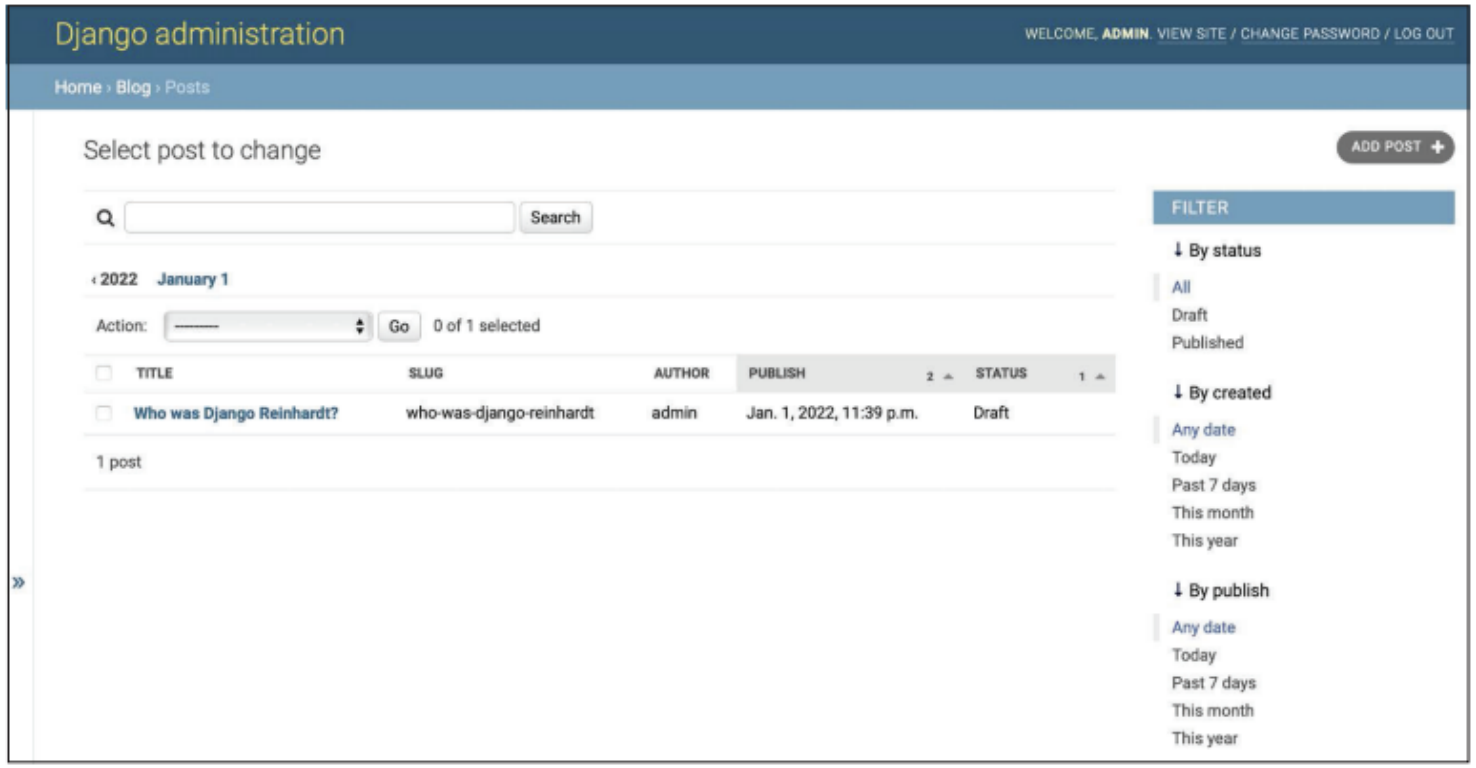
Давайте адаптируем модель **admin**, внеся в нее еще несколько опций. Отредактируйте файл **admin.py** приложения **blog**, изменив его, как показано ниже. Новые строки выделены жирным шрифтом:

```
from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
    list_filter = ['status', 'created', 'publish', 'author']
    search_fields = ['title', 'body']
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ['author']
    date_hierarchy = 'publish'
    ordering = ['status', 'publish']
```

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Вернитесь в свой браузер и перезагрузите страницу списка постов. Теперь она будет выглядеть примерно так:



Конкретно-прикладное представление списка на сайте администрирования для модели Post

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Вы видите, что отображаемые на странице списка постов поля соответствуют тем, которые мы указали в атрибуте **list_display**.

Теперь страница списка содержит правую боковую панель, которая позволяет фильтровать результаты по полям, включенным в атрибут **list_filter**. На странице появилась строка поиска. Это вызвано тем, что мы определили список полей, по которым можно выполнять поиск, используя атрибут **search_fields**.

Чуть ниже строки поиска находятся навигационные ссылки для навигации по иерархии дат; это определено атрибутом **date_hierarchy**.

Вы также видите, что по умолчанию посты упорядочены по столбцам **STATUS** (Статус) и **PUBLISH** (Опубликован). С помощью атрибута **ordering** были заданы критерии сортировки, которые будут использоваться по умолчанию

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Далее кликните по ссылке **ADD POST** (Добавить пост). Здесь вы тоже заметите некоторые изменения. При вводе заголовка нового поста поле **slug** заполняется автоматически.

Сообщили Django, что нужно предзаполнять поле **slug** данными, вводимыми в поле **title**, используя атрибут **prepopulated_fields**:

Теперь модель **slug** автоматически предзаполняется при наборе заголовка на клавиатуре

АДАПТАЦИЯ ВНЕШНЕГО ВИДА МОДЕЛЕЙ ПОД КОНКРЕТНО-ПРИКЛАДНУЮ ЗАДАЧУ

Кроме того, теперь поле author отображается поисковым виджетом, который будет более приемлемым, чем выбор из выпадающего списка, когда у вас тысячи пользователей. Это достигается с помощью атрибута raw_id_fields и выглядит следующим образом

Виджет для отбора ассоциированных объектов для поля author модели Post

Всего несколькими строками исходного кода мы адаптировали отображение модели на сайте администрирования. Адаптировать и расширять сайт администрирования можно огромным числом способов

РАБОТА С НАБОРАМИ ЗАПРОСОВ QUERYSET И МЕНЕДЖЕРАМИ

Теперь, когда у нас есть полнофункциональный сайт администрирования, чтобы управлять постами блога, самое время научиться программно читать контент из базы данных и писать его в базу данных

Встроенный в Django объектно-реляционный преобразователь **ORM** (object-relational mapper) – это мощный **API** абстракции базы данных, который позволяет легко создавать, извлекать, обновлять и удалять объекты. ORM-преобразователь дает возможность генерировать запросы на языке **SQL**, используя объектно-ориентированную парадигму Python. Его можно трактовать как способ взаимодействия с базой данных в Python'овском стиле вместо написания сырых **SQL-запросов**.

РАБОТА С НАБОРАМИ ЗАПРОСОВ QUERYSET И МЕНЕДЖЕРАМИ

ORM-преобразователь соотносит модели с таблицами базы данных и предоставляет простой Python'овский интерфейс взаимодействия с базой данных. ORM-преобразователь генерирует SQL-запросы и соотносит результаты с объектами модели. ORM-преобразователь совместим с реляционными системами управления базами данных **MySQL, PostgreSQL, SQLite, Oracle** и **MariaDB**.

Напомним, что базу данных своего проекта можно определять в настроечном параметре DATABASES файла **settings.py** проекта. Django может работать с несколькими базами данных одновременно, при этом можно программировать маршрутизаторы баз данных, чтобы создавать конкретно-прикладные схемы маршрутизации данных. После создания своих моделей данных Django предоставит бесплатный **API** для взаимодействия с ними

Встроенный в Django ORM-преобразователь основан на итерируемых наборах запросов **QuerySet**. Итерируемый набор запросов **QuerySet** – это коллекция запросов к базе данных, предназначенных для извлечения объектов из базы данных. К наборам запросов можно применять фильтры, чтобы сужать результаты запросов на основе заданных параметров.

СОЗДАНИЕ ОБЪЕКТОВ

Открыть оболочку Python:

```
python manage.py shell
```

наберите следующие ниже строки:

```
>> from blog.models import Post
>> user = User.objects.get(username='admin')
>> post = Post(title='Another post',
>>             slug='another-post',
>>             body='Post body.',
>>             author=user)
>> post.save()
```

Сначала извлекается объект **user** с пользовательским именем **admin**.

Метод **get()** позволяет извлекать из базы данных только один объект. Обратите внимание, что этот метод ожидает результат, совпадающий с запросом. Если база данных не возвращает результатов, то указанный метод вызовет исключение **DoesNotExist**, а если база данных возвращает более одного результата, то он вызовет исключение **MultipleObjectsReturned**.

Оба исключения являются атрибутами модельного класса, на котором выполняется запрос. Затем создается экземпляр класса **Post** с конкретно-прикладным заголовком, слагом и телом и задаем пользователя, которого мы ранее извлекли, в качестве автора поста

СОЗДАНИЕ ОБЪЕКТОВ

Этот объект находится в памяти и не сохраняется в базе данных;

Мы создали объект Python, который можно использовать на стадии работы программы, но который не сохраняется в базе данных. Наконец, мы сохраняем объект **Post** в базе данных, используя метод **save()**

Приведенное выше действие за кулисами выполняет инструкцию **SQL INSERT**. Сначала мы создали объект в памяти, а затем сохранили его в базе данных. Создавать объект и сохранять его в базе данных также можно одной операцией, используя метод **create()**. Это делается следующим образом:

```
Post.objects.create(title='One more post',
                    slug='one-more-post',
                    body='Post body.',
                    author=user)
```

ОБНОВЛЕНИЕ ОБЪЕКТОВ

Теперь измените заголовок поста на что-то другое и снова сохраните объект:

```
>>> post.title = 'New title'
>>> post.save()
```

На этот раз метод save() исполняет инструкцию SQL UPDATE

ИЗВЛЕЧЕНИЕ ОБЪЕКТОВ

Одиночный объект извлекается из базы данных методом **get()**. Мы применили этот метод посредством метода **Post.objects.get()**.

Каждая модель Django имеет по меньшей мере один модельный менеджер, а менеджер, который применяется по умолчанию, называется **objects**.

Набор запросов **QuerySet** можно получать с помощью модельного менеджера. Для того чтобы извлечь все объекты из таблицы, используется метод **all()** применяемого по умолчанию менеджера **objects**. Например:

```
>>> all_posts = Post.objects.all()
```

ИЗВЛЕЧЕНИЕ ОБЪЕКТОВ

Вот как мы создаем набор запросов **QuerySet**, который возвращает все объекты базы данных. Обратите внимание, что этот **QuerySet** еще не исполнен.

Наборы запросов **QuerySet** в Django являются ленивыми, то есть они вычисляются только тогда, когда это приходится делать. Подобное поведение придает наборам запросов **QuerySet** большую эффективность.

Если не назначать набор запросов **QuerySet** переменной, а вместо этого писать его непосредственно в оболочке **Python**, то инструкция **SQL** набора запросов будет исполняться, потому что вы побуждаете ее генерировать результат:

```
Post.objects.all()
<QuerySet [<Post: Who was Django Reinhardt?>, <Post: New title>]>
```

ПРИМЕНЕНИЕ МЕТОДА FILTER()

Для фильтрации набора запросов **QuerySet** можно использовать метод **filter()** менеджера. Например, все посты, опубликованные в 2022 году, можно получить, используя следующий ниже набор запросов:

```
>>> Post.objects.filter(publish__year=2022)
```

Фильтрация также может выполняться по нескольким полям. Например, все посты, опубликованные в 2022 году автором с пользовательским именем admin, можно получить следующим образом:

```
>>> Post.objects.filter(publish__year=2022, author__username='admin')
```

Это приравнивается к формированию одного и того же набора запросов **QuerySet**, соединяющего несколько фильтров в цепочку:

```
>>> Post.objects.filter(publish__year=2022) \
>>>             .filter(author__username='admin')
```

ПРИМЕНЕНИЕ МЕТОДА EXCLUDE()

Определенные результаты можно исключать из набора запросов **QuerySet**, используя метод **exclude()** менеджера.

Например, все посты, опубликованные в 2022 году, заголовки которых не начинаются со слова **Why (Почему)**, можно получить следующим образом:

```
>>> Post.objects.filter(publish__year=2022) \
>>>              .exclude(title__startswith='Why')
```

ПРИМЕНЕНИЕ МЕТОДА ORDER_BY()

Используя метод **order_by()** менеджера, можно упорядочивать результаты по разным полям. Например, можно извлечь все объекты, упорядоченные по их полю **title**, как показано ниже:

```
>>> Post.objects.order_by('title')
```

Подразумевается возрастающий порядок. Убывающий порядок указывается с помощью префикса с отрицательным знаком. Например:

```
>>> Post.objects.order_by('-title')
```

УДАЛЕНИЕ ОБЪЕКТОВ

Если необходимо удалить объект, то это можно сделать из экземпляра объекта, используя метод delete():

```
>>> post = Post.objects.get(id=1)
>>> post.delete()
```

Обратите внимание, что удаление объектов также приводит к удалению любых зависимых взаимосвязей объектов **ForeignKey**, в случае если параметр **on_delete** задан равным значению **CASCADE**.

КОГДА ВЫЧИСЛЯЮТСЯ НАБОРЫ ЗАПРОСОВ QUERYSET

Создание набора запросов **QuerySet** не требует каких-либо действий с базой данных до тех пор, пока он не будет вычислен. Наборы запросов обычно возвращают еще один невычисленный набор запросов. В наборе запросов можно конкатенировать столько фильтров, сколько потребуется, и база данных не будет затронута до тех пор, пока набор запросов не будет вычислен.

При вычислении набора запросов он конвертируется в запрос на языке **SQL** к базе данных. Наборы запросов **QuerySet** вычисляются только в следующих ниже случаях:

- при первом их прокручивании в цикле;
- при их нарезке, например **Post.objects.all()[:3]**;
- при их консервации в поток байтов или кешировании;
- при вызове на них функций **repr()** или **len()**;
- при вызове на них функции **list()** в явной форме;
- при их проверке в операциях **bool(), or, and** или **if**

СОЗДАНИЕ МОДЕЛЬНЫХ МЕНЕДЖЕРОВ

По умолчанию в каждой модели используется менеджер **objects**. Этот менеджер извлекает все объекты из базы данных. Однако имеется возможность определять конкретно-прикладные модельные менеджеры.

Давайте создадим конкретно-прикладной менеджер, чтобы извлекать все посты, имеющие статус **PUBLISHED**.

Есть два способа добавлять или адаптировать модельные менеджеры под конкретно-прикладную задачу: можно добавлять дополнительные методы менеджера в существующий менеджер либо создавать новый менеджер, видоизменив изначальный набор запросов **QuerySet**, возвращаемый менеджером.

Первый метод предоставляет обозначение набора запросов в виде **Post**. **objects.my_manager()**, а второй предоставляет обозначение набора запросов в виде **Post.my_manager.all()**.

СОЗДАНИЕ МОДЕЛЬНЫХ МЕНЕДЖЕРОВ

Мы выберем второй метод, чтобы реализовать менеджер, который позволит извлекать посты, используя обозначение **Post.published.all()**.

Отредактируйте файл **models.py** приложения **blog**, добавив конкретно-прикладной менеджер, как показано ниже. Новые строки выделены жирным шрифтом:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    # поля модели
    # ...

    objects = models.Manager() # менеджер, применяемый по умолчанию
    published = PublishedManager() # конкретно-прикладной менеджер

    class Meta:
        ordering = ['-publish']

    def __str__(self):
        return self.title
```

СОЗДАНИЕ МОДЕЛЬНЫХ МЕНЕДЖЕРОВ

Первый объявленный в модели менеджер становится менеджером, который используется по умолчанию. Для того чтобы указать другой такой менеджер, применяется Meta-атрибут **default_manager_name**.

Если менеджер в модели не определен, то Django автоматически создает для нее стандартный менеджер **objects**. Если в своей модели вы объявляете какие-либо менеджеры, но также хотите сохранить менеджер **objects**, то вы должны добавить его в свою модель явным образом. В приведенном выше исходном коде мы добавили в модель Post стандартный менеджер **objects** и конкретно-прикладной менеджер **published**.

СОЗДАНИЕ МОДЕЛЬНЫХ МЕНЕДЖЕРОВ

Метод **get_queryset()** менеджера возвращает набор запросов **QuerySet**, который будет исполнен. Мы переопределили этот метод, чтобы сформировать конкретно-прикладной набор запросов **QuerySet**, фильтрующий посты по их статусу и возвращающий поочередный набор запросов **QuerySet**, содержащий посты только со статусом **PUBLISHED**.

Теперь, когда мы определили для модели **Post** конкретно-прикладной менеджер, давайте его протестируем! Следующей ниже командой снова запустите сервер разработки из командной оболочки:

```
python manage.py shell
```

СОЗДАНИЕ МОДЕЛЬНЫХ МЕНЕДЖЕРОВ

Теперь можно импортировать модель **Post** и извлечь все опубликованные посты, заголовки которых начинаются с **Who**, исполнив следующий ниже набор запросов **QuerySet**:

```
>>> from blog.models import Post
>>> Post.published.filter(title__startswith='Who')
```

Для того чтобы получить результаты этого набора запросов, проверьте, чтобы поле **status** было равным значению **PUBLISHED** в объекте **Post**, поле **title** которого начинается со слова **Who**.