

# Занятие #45. Модели, админка, django-shell

## Модели

В Django **модель** - это специальный класс, который представляет ваши данные. Модель не является непосредственно данными - это интерфейс доступа к данным. Обычно каждая модель соответствует одной сущности или таблице в базе данных. Каждый объект модели представляет одну запись (существующую или новую) в своей таблице.

Модели в Django принято писать в файле `models.py`. Он свой в каждом приложении и именно в нём Django и другие программисты будут искать ваши модели.

Все модели в Django наследуются либо от базового класса `Model`, который импортируется из пакета `django.db.models` либо друг от друга.

Например, на прошлом занятии мы научились вводить данные для статей, но пока они нигде не сохраняются, мы можем только увидеть заготовку будущей статьи на странице предпросмотра. Чтобы иметь возможность сохранять статьи, нужно описать их сущность в виде модели. Добавьте следующий код в файл `webapp/models.py`:

```
class Article(models.Model):
    title = models.CharField(max_length=200, null=False,
blank=False, verbose_name='Заголовок')
    text = models.TextField(max_length=3000, null=False,
blank=False, verbose_name='Текст')
    author = models.CharField(max_length=40, null=False,
blank=False, default='Unknown', verbose_name='Автор')
    created_at = models.DateTimeField(auto_now_add=True,
verbose_name='Время создания')
    updated_at = models.DateTimeField(auto_now=True,
verbose_name='Время изменения')

    def __str__(self):
        return "{}. {}".format(self.pk, self.title)
```

Здесь мы описали модель `Article` - статья, аналогично описанию сущности для создания таблицы в базе данных. Атрибуты класса `Article` - `title`, `text`, `author`, `created_at`, `updated_at` соответствуют полям в будущей таблице. Такие же атрибуты будет иметь каждый объект класса `Article`, представляющий отдельную статью.

Тип поля определяется тем классом, к которому относится значение каждого из этих атрибутов:

- `CharField` - строковый тип, соответствует типу `varchar` в базе данных,
- `TextField` - текстовый тип, соответствует типам `varchar` или `text` в базе данных,
- `DateTimeField` - дата и время, соответствует типу `datetime` в базе данных.

Первичный ключ-код в моделях Django обычно не пишется, т.к. Django добавляет его автоматически под названием `id`. Также у всех объектов модели есть свойство `pk` - ссылка на первичный ключ модели, которая всегда указывает на него, даже если он отличается от поля `id`.

Каждый тип поля имеет свой набор свойств, переданных в качестве аргументов при создании нового поля:

- `max_length` - размер поля в базе данных. Применяется в основном для текстовых полей.
- `null` - является ли поле обязательным в базе данных, может ли быть пустым
- `blank` - является ли поле обязательным при заполнении в формах. Это значение не влияет на базу данных, но влияет на *валидацию* данных в формах Django.
- `default` - значение для поля по умолчанию, если оно пустое.
- `auto_now_add` - специальное свойство для полей типа `Date` и `DateTime`. Если включено, то при добавлении записи Django автоматически запишет в это поле текущую дату и время.
- `auto_now` - аналогично полю `auto_now_add`, но сохраняет текущую дату и время каждый раз при сохранении записи (при добавлении и при редактировании).

Также существуют и другие классы для различных типов полей данных. О них можно прочитать по ссылке в конце раздатки. Также мы будем применять некоторые из них позже.

Метод `__str__()` у модели необходим для представления объектов модели в читаемом виде при выводе в админ-панели или в терминале. В данном примере мы будем выводить ключ и заголовок статьи.

## Миграции

Модель не является таблицей в базе данных, а только определяет структуру, соответствующую возможной таблице. Чтобы создать таблицу на основе модели в Django нужно создать **миграцию**. **Миграции** описывают изменения в базе данных: создание и удаление таблиц, создание, изменение и удаление полей в них, создание и удаление связей между таблицами, изменение общей информации о моделях (метаданных) и т.д. Миграции обычно связаны с определёнными приложениями. Таким образом, каждое приложение может иметь независимую структуру таблиц в базе данных (и даже подключаться к разным базам) и свой набор миграций, что позволяет легко подключать и отключать приложения в проекте.

Чтобы создать миграцию используется команда

```
./manage.py makemigrations приложение_1 приложение_2 приложение_3
...
```

Вы можете перечислить несколько приложений через пробел. Если вы не укажете приложение, то Django попытается создать миграции для каждого приложения, где описание моделей в файле `models.py` отличается от реальной структуры базы данных.

Первая миграция называется `initial`. Давайте создадим миграцию для создания таблицы для модели `Article` из приложения `webapp`:

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

После создания миграции в папке `webapp/migrations` вы увидите миграцию с названием `0001_initial.py` или аналогичным. В названии миграции `0001` - это её

номер, на который ориентируется Django, чтобы определить последовательность новых миграций.

Если вы файл миграции, откроете, то увидите следующий код:

```
# Generated by Django 2.1.3 on 2018-11-22 12:00

from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            ... # описание модели Article.
        ),
    ]
```

Первая строчка указывает на то, что этот файл был автоматически сгенерирован Django. В большинстве случаев вам не нужно редактировать миграции, т.к. Django уже прописал здесь весь необходимый код. Ссылки на другие миграции, которые необходимы для применения этой, записаны в свойстве `dependencies`, а действия, которые совершает в базе текущая миграция - в `operations`.

Если вы создали миграцию по ошибке или забыли что-то в неё добавить (ещё какие-то изменения в структуре данных), на этом этапе, до применения миграции, вы ещё можете её удалить или поменять.

После создания миграции её нужно **применить**. **Применение** миграции означает реальное создание в базе данных нужных таблиц/полей или внесение других изменение в базу данных **физически**.

Применение миграций выполняется командой

```
./manage.py migrate приложение_1 приложение_2 приложение_3 ...
```

Если вы не укажете ни одного названия приложения, Django применит все миграции, которые не были применены ранее. Django отмечает в специальной таблице в базе данных, какие миграции он уже применил, чтобы не повторять их снова.

Примените миграции командой

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

**После применения миграций не переименовывайте и не удаляйте их**, т.к. последующие миграции зависят от предыдущих, а название миграции записано в базе данных. Нарушение этого правила может привести к тому, что ваши миграции перестанут применяться.

**Никогда не редактируйте содержимое старых миграций**, т.к. структура базы данных от этого не изменится, но ваши модели не будут соответствовать структуре данных в БД. При этом вы не сможете применить "исправленные" миграции повторно, т.к. у Django записано, что он их уже применил. **В худшем случае вы можете потерять ваши данные при попытке починить миграции и базу!**

Когда у вас меняется структура моделей - добавляются новые модели и поля или удаляются старые, или меняются их свойства, каждый раз **создавайте и применяйте новые миграции**.

Также Django позволяет вам откатывать (отменять) миграции, применённые по ошибке. Чтобы откатить миграции, укажите команде migrate номер той миграции, до которой вы хотите совершить откат. Например, если вы находитесь на миграции 0002, то вернуться к миграции 0001 можно следующей командой:

```
./manage.py migrate webapp 0001
```

Тем не менее, если миграция добавила в базу какие-то поля, которые после этого были заполнены данными, то откат грозит вам потерей этих данных. Если миграция удаляет из базы какие-то поля, то при откате данные из этих полей не восстановятся, и придётся загружать их из резервной копии.

Чтобы откатить начальную миграцию, 0001, нужно вызвать команду migrate с параметром zero:

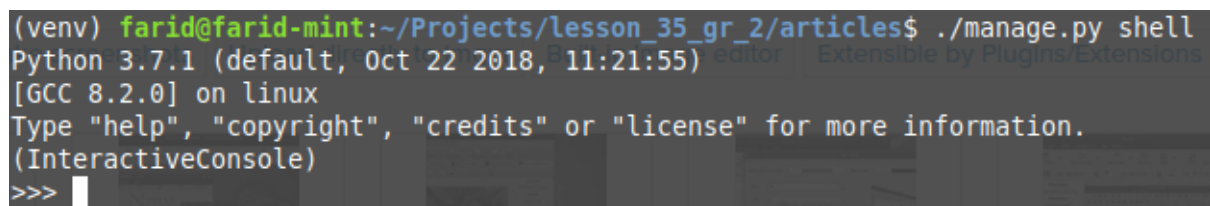
```
./manage.py migrate webapp zero
```

Т.к. миграции с номером 0000 не существует.

## Django-консоль

Теперь, когда у нас есть таблица в базе данных для модели Article мы можем попробовать добавить в неё какие-то данные или посмотреть, что в ней есть. Прежде, чем писать код на сайте, откройте django-консоль (django-shell) командой

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



```
(venv) farid@farid-mint:~/Projects/lesson_35_gr_2/articles$ ./manage.py shell
Python 3.7.1 (default, Oct 22 2018, 11:21:55)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Вы окажетесь в консоли, аналогичной консоли питона, но с корректно подключенным кодом из проекта, в котором находитесь. Здесь доступны ваши модели, представления и другой код вашего приложения на Django. Импортируйте модель Article:

```
>>> from webapp.models import Article
```

Для доступа к данным через модель используется специальный объект-менеджер. Он является атрибутом класса Articles (и других классов моделей), называется objects и содержит методы для выполнения запросов к данным - для получения данных, для добавления данных, для редактирования и удаления данных.

Например, вы можете получить список всех статей методом `all()`:

```
>>> Article.objects.all()
<QuerySet []>
```

Сейчас у нас нет ни одной статьи в базе данных, поэтому метод `all()` вернул пустой список, или точнее - объект `QuerySet`, о которых мы поговорим позже. Пока считайте его списком.

Попробуем добавить несколько статей:

```
>>> Article.objects.create(author='Name', title='Title',
text='Some long informative text')
<Article: 1. Title>
```

Другой способ создать статью:

```
new_article = Article()
new_article.author = 'Name 2'
new_article.title = 'Title 2'
new_article.text = 'Some long informative text 2'
new_article.save()
```

Здесь мы создали новый объект класса `Article`, назначили ему значения атрибутов, и сохранили методом `save()`. Попробуем теперь посмотреть, какие у нас есть статьи в базе:

```
>>> Article.objects.all()
<QuerySet [<Article: 1. Title>, <Article: 2. Title 2>]>
```

Возьмём, например, первую статью:

```
article = Article.objects.all()[0]
```

Атрибуты объекта `article` соответствуют полям в базе данных, а их значения - это значения из базы:

```
>>> article.title
'Title'
>>> article.text
'Some long informative text'
>>> article.author
'Name'
```

Также у каждого объекта каждой модели есть специальный атрибут `pk`, который соответствует первичному ключу записи, представляемой этой моделью.

```
>>> article.pk
1
```

Для редактирования объекта модели вы можете менять значения его атрибутов:

```
>>> article.text = 'Another long informative text'
```

Для сохранения данных после редактирования объекта модели также применяется метод `save()`:

```
>>> article.save()
```

Проверим, что данные успешно изменились - "пере-запросим" объект статьи из базы данных. Не будем вызывать `all()` на этот раз, т.к. нам нужен всего один объект, и мы знаем его ключ. Здесь можно применить метод `get()`, который используется для получения одной записи:

```
>>> article = Article.objects.get(pk=1)
```

Метод `get()` всегда возвращает только один подходящий объект и выкидывает ошибку, если такого объекта не существует или найдено несколько подходящих объектов.

Проверим, что текст изменился:

```
>>> article.text
'Another long informative text'
```

Для фильтрации более, чем одного объекта, нужно применять метод `filter()`, который принимает поля и их значения в виде ключевых параметров, и возвращает все подходящие объекты либо пустой список (кваерисет).

Для примера создадим ещё одну статью от автора "Name" с другим заголовком и содержанием:

```
>>> Article.objects.create(author='Name', title='New Title',
text='New Text')
<Article: 3. New Title>
```

Теперь применим фильтр по автору:

```
>>> Article.objects.filter(author='Name')
<QuerySet [<Article: 1. Title>, <Article: 3. New Title>]>
```

Как видим, вернулись две статьи - обе статьи этого автора.

Сузим критерии поиска, добавим фильтр по названию (заголовку) статьи:

```
>>> Article.objects.filter(author='Name', title='New Title')
<QuerySet [<Article: 3. New Title>]>
```

Видим, что теперь возвращается только третья статья, у которой совпадают и заголовок, и автор.

Пример пустых результатов поиска, если ни один объект не подходит:

```
>>> Article.objects.filter(author='No')
<QuerySet []>
```

Удалить объект модели можно с помощью метода `delete()`. Сначала найдём его:

```
>>> article = Article.objects.get(pk=1)
```

Потом удалим:

```
>>> article.delete()
```

Проверим, что он действительно удалён, запросив все статьи из базы:

```
>>> Article.objects.all()
<QuerySet [<Article: 2. Title>, <Article: 3. New Title>]>
```

Запись из базы действительно удалена, но объект `article` со всеми его свойствами всё ещё доступен в коде программы и находится в оперативной памяти. Пока код в текущей области видимости не выполнен до конца, мы всё ещё можем сохранить объект `article` обратно в базу:

```
>>> article.save()
```

И проверим, что он снова появился:

```
>>> Article.objects.all()
<QuerySet [<Article: 1. Title>, <Article: 2. Title 2>, <Article: 3. New Title>]>
```

(Обычно удалённые объекты обратно не сохраняют, этот код приведён только для примера, чтобы показать, что объект модели в коде и запись в БД существуют независимо друг от друга.)

Последнее, что вам достаточно сейчас знать - это сортировка объектов. Она выполняется методом `order_by()` на квериcетax. Например, отсортируем статьи по заголовку:

```
>>> Article.objects.all().order_by('title')
<QuerySet [<Article: 3. New Title>, <Article: 1. Title>, <Article: 2. Title 2>]>
```

Третья статья выходит выше.

Для сортировки в обратном порядке, по убыванию, перед названием поля ставится знак "минус":

```
>>> Article.objects.all().order_by('-title')
<QuerySet [<Article: 2. Title 2>, <Article: 1. Title>, <Article: 3. New Title>]>
```

Также при необходимости можно указать в сортировке несколько полей. Например, одновременная сортировка по заголовку и имени автора могла бы выглядеть так:

```
>>> Article.objects.all().order_by('title', 'author')
```

## Интерфейс администратора

Итак, у нас есть модель и соответствующая ей таблица в базе данных. Давайте попробуем поработать с этой моделью из встроенного в Django интерфейса администратора - или "админки".

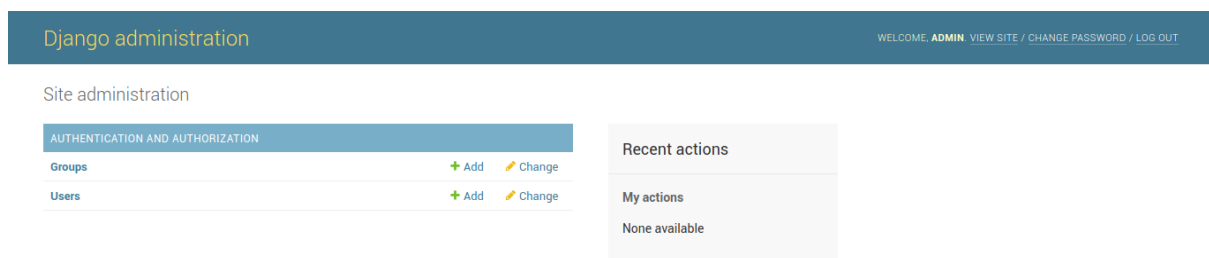
Сначала вам нужно создать суперпользователя - администратора, который будет иметь доступ в админку и ко всем данным (моделям, таблицам).

Суперпользователь создаётся командой

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

Вам нужно ввести логин и пароль для будущего суперпользователя, и по желанию - email. Далее вы можете использовать эти логин и пароль для входа в админку, которая находится по пути на сайте /admin.

Запустите django-сервер на порту 8000 и откройте в браузере страницу <http://localhost:8000/admin>. Войдите с логином и паролем ранее созданного суперпользователя. Здесь вы увидите список всех моделей, имеющих в приложениях вашего сайта. Но здесь нет модели articles:



Чтобы модель Articles появилась в админке, вам нужно **зарегистрировать** её в файле конфигурации admin.py. Такой файл имеется в каждом приложении django, и используется для настройки отображения различных моделей в интерфейсе администратора. Пока просто добавим в него нашу модель Article. Для этого импортируйте её в файл webapp/admin.py и добавьте туда строчку

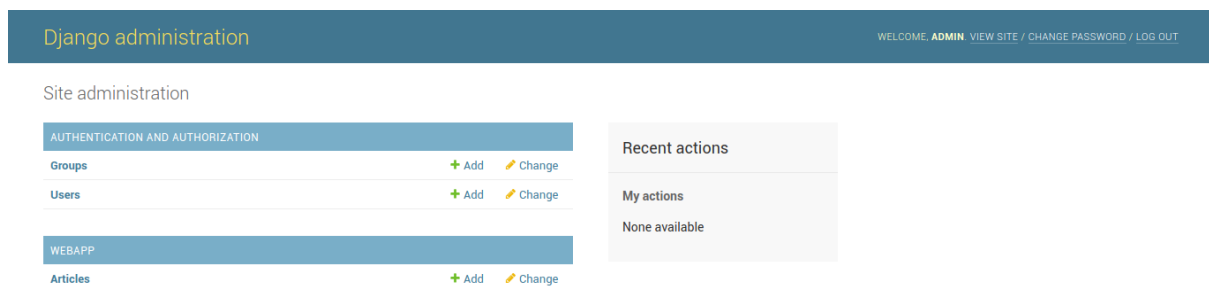
```
admin.site.register(Article)
```

Полный код файла webapp/admin.py:

```
from django.contrib import admin
from webapp.models import Article      # импортируем модель

admin.site.register(Article)
```

Теперь вы можете просматривать данные модели Article, добавлять новые статьи и редактировать их из админ-панели.





Админ-панель позволяет вам управлять всеми данными в базе проекта, а также создавать и удалять пользователей, группы и выдавать им разрешения на доступ к разным данным.

Админ-панель можно настраивать. Для каждой модели вы можете создать специальный класс-"админ" на базе класса `ModelAdmin` из модуля `django.contrib.admin`, который будет содержать конфигурацию админ-панели для этой модели. Например, для статей можно написать такой админ-класс:

```
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['id', 'title', 'author', 'created_at']
    list_filter = ['author']
    search_fields = ['title', 'text']
    fields = ['title', 'author', 'text', 'created_at',
'updated_at']
    readonly_fields = ['created_at', 'updated_at']
```

Его свойства:

- `list_display` - список полей, которые будут выводиться в списке объектов этой модели. Если задано это свойство, список объектов модели в админке будет выводиться в виде таблицы с сортировкой по каждому полю (нужно кликнуть заголовок, чтобы активировать сортировку).
- `list_filter` - список полей, по которым можно отфильтровать записи. Поля и их возможные значения выводятся в виде ссылок в правой части страницы списка объектов модели. Здесь обычно указывают поля, имеющие небольшой набор значений, на каждое из которых приходится по несколько записей. Например, у нас это - автор статьи.
- `search_fields` - если задано это свойство, админ-панель выведет форму поиска на странице списка объектов этой модели. Поиск будет выполняться по вхождению искомого значения поиска в значения указанных в этом свойстве полей.
- `fields` - свойство, в котором перечисляются все поля модели, доступные в админ-панели при просмотре одного объекта и в форме создания объекта. Сейчас здесь выводятся все поля. `id` указывать не нужно, т.к. в самой модели это поле не прописано и редактировать его нельзя. Можно указать в качестве поля имя метода без аргументов, который возвращает подходящее для вывода значение, например, `"__str__"`.
- `exclude` - альтернатива свойству `fields`, список тех полей, которые не нужно выводить в админ-панели. Если задано свойство `exclude`, будут выводиться все поля, которые в нём не указаны. Для вывода всех полей `exclude` можно указать равным пустому списку: `exclude = []`. `fields` и `exclude` - взаимоисключающие, поэтому одновременно можно задавать только что-то одно из них.
- `readonly_fields` - поля "только для чтения". Значения этих полей могут выводиться на странице просмотра и редактирования, но менять их нельзя. Также они не выводятся в форме создания объектов модели. Здесь это свойство необходимо, т.к. мы хотим вывести даты создания и последнего обновления статьи, но из-за авто-заполнения (`auto_now` и `auto_now_add`) менять вручную их нельзя, поэтому требуется их обозначить, как `"readonly"`.

Для применения настроек из админ-класса, нужно указать его вторым аргументом при регистрации модели:

```
admin.site.register(Article, ArticleAdmin)
```

## Итоговый вид админ-панели для модели Article:

Django administration

WELCOME, АДМИН. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Webapp > Articles

Select article to change

ADD ARTICLE +

Q  Search

Action:  Go 0 of 6 selected

ID	ЗАГОЛОВОК	АВТОР	ВРЕМЯ СОЗДАНИЯ
<input type="checkbox"/> 7	New Article	Me	Sept. 2, 2019, 9:28 p.m.
<input type="checkbox"/> 6	Test	He	Sept. 2, 2019, 8:43 p.m.
<input type="checkbox"/> 5	Even More	Me	Sept. 2, 2019, 8:07 p.m.
<input type="checkbox"/> 4	Article 3	She	Sept. 2, 2019, 8:03 p.m.
<input type="checkbox"/> 3	Title	He	Sept. 2, 2019, 8:02 p.m.
<input type="checkbox"/> 2	Article	Me	Sept. 2, 2019, 8:17 p.m.

6 articles

FILTER

By Автор

- All
- He
- Me
- She

Django administration

WELCOME, АДМИН. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Webapp > Articles > Article

Change article

HISTORY

Заголовок:

Автор:

Текст:

Время создания: Sept. 2, 2019, 8:17 p.m.

Время изменения: Sept. 2, 2019, 8:17 p.m.

Delete Save and add another Save and continue editing SAVE

Подробнее по настройке админ-панели и свойствам админ-классов см. по ссылке в конце раздатки.

## Фикстуры

**Фикстуры** - это небольшие файлы в формате json или yaml, которые используются для хранения и перемещения данных из базы в сериализованном виде.

Фикстуры Django могут хранить только данные, и не описывают схему БД (таблицы и связи), в отличие от дампов SQL.

Фикстуры можно использовать, как резервные копии, из которых можно восстановить базу данных, или создать её копию.

При восстановлении фикстуры перезаписывают данные с совпадающими id, и таким образом могут затереть существующие данные, поэтому их обычно используют на

локальных/тестовых БД или для загрузки в базу начальных данных при первом запуске проекта.

Для создания фикстур применяется команда:

```
./manage.py dumpdata
```

Без параметров она выводит дамп данных всех приложений проекта в консоль.

Если вы хотите добавить в дамп конкретные приложения или модели, вы можете указать их в качестве аргументов этой команды. Например, команда, которая создаёт дамп только с данными из моделей приложения `webapp` и модели `User` (пользователь) из встроенного приложения `auth`:

```
./manage.py dumpdata webapp auth.user
```

`webapp` - обозначает приложение `webapp`.

`auth.user` - обозначает модель `User` из приложения `auth`.

В общем случае имя каждой модели пишется после имени приложения, в нижнем регистре, и отделяется от имени своего приложения точкой.

По умолчанию дамп выводится в одну строчку. Чтобы сделать вывод более красивым, вы можете применить параметр `--indent`, который сообщает Django выводить дамп с переносами и отступами. Например, для выравнивания в два пробела:

```
./manage.py dumpdata --indent=2      # "indent=2" пишется без пробелов!  
./manage.py dumpdata --indent=2 webapp auth.user # только webapp и User
```

По умолчанию Django ищет фикстуры в папках **fixtures** внутри приложений. Например, вы можете создать папку **webapp/fixtures** и хранить фикстуры там, тогда Django сам найдёт их при загрузке и вам не понадобится указывать к ним путь.

Теперь пришло время соединить всё это вместе:

```
mkdir webapp/fixtures  
./manage.py dumpdata --indent=2 webapp auth.user >  
webapp/fixtures/dump.json
```

Теперь у вас есть дамп вашей базы. Проверить его содержимое можно командой `cat`:

```
cat webapp/fixtures/dump.json
```

Вы можете иметь несколько дампов одновременно - по одному для каждого приложения/модели, разные дампы для разных задач и т.д.

Восстановление фикстур выполняется командой:

```
./manage.py loaddata
```

По умолчанию она загружает в базу все фикстуры из папок `fixtures` в приложениях. Если вы хотите загрузить фикстуру из другого пути, нужно либо добавить его в

настройку `FIXTURE_DIRS` в `settings.py` (список путей к фикстурам), либо указать путь к фикстуре явно:

```
./manage.py loaddata other_fixture.json
```

Таким же образом можно загрузить только одну или несколько фикстур, если вы не хотите загружать все фикстуры разом.

Чтобы не было ошибок, на момент загрузки фикстуры база данных уже должна существовать, и иметь такую же структуру, как и данные в фикстуре, поэтому не забывайте сначала мигрировать базу.

Также обновляйте существующие фикстуры каждый раз, когда вы применяете (после их создания) новые миграции, чтобы фикстуры были актуальны.

## Список объектов

Теперь у нас есть статьи в базе. Как их вывести на какую-либо страницу? Для начала попробуем вывести список статей на главной. Для этого нужно получить к ним доступ в соответствующем представлении. Откройте файл **`webapp/views.py`** и добавьте в его начало импорт модели `Article`:

```
from webapp.models import Article
```

Далее замените код представления `index_view` на следующий, чтобы передать список всех статей в контексте для отображения шаблона.:

```
def index_view(request):
    articles = Article.objects.all()
    context = {
        'articles': articles
    }
    return render(request, 'index.html', context)
```

Замените код в `body` шаблона `index.html` на следующий:

```
<h1>Articles</h1>
{% for article in articles %}
    <h2>{{ article.title }}</h2>
    <p><a href="#">Подробнее...</a></p>
    <hr>
{% endfor %}
```

Теги `{% for %}` и `{% endfor %}` обозначают генерацию содержимого страницы в цикле. Тег `{% for %}` обозначает начало повторяющейся разметки, а тег `{% endfor %}` - конец. В данном случае мы таким образом выводим подряд все заголовки статей и ссылки на продолжение.

Пока эти ссылки никуда не ведут. Давайте добавим страницу для просмотра статьи:

Добавьте представление **`article_view`** в **`webapp/views.py`**:

```
def article_view(request):
    article_id = request.GET.get('pk')
    article = Article.objects.get(pk=article_id)
```

```
context = {'article': article}
return render(request, 'article_view.html', context)
```

Это представление ожидает, что в параметрах запроса будет передан номер - код статьи в базе данных.

Далее добавьте ссылку на него в `urls.py`:

```
urlpatterns = [
    ...
    path('article/', article_view)
]
```

Не забудьте импортировать представление `article_view` в `urls.py`!

Далее исправьте шаблон **article\_view.html** таким образом, чтобы он отображал данные из объекта `article` вместо переменных `title`, `content` и `author`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>View</title>
</head>
<body>
    <p><a href="/">На главную</a></p>
    <h1>Article:</h1>
    <h2>{{ article.title }}</h2>
    <p>{{ article.text }}</p>
    <p>By: {{ article.author }}</p>
</body>
</html>
```

К свойствам переменных в шаблонах можно обращаться так же, как и в коде - через точку ("."). Таким образом можно обращаться только к свойствам, **нельзя вызывать метод**. Также можно подобным образом обращаться к индексам списков и ключам словарей, через точку вместо квадратных скобок:

```
{{ my_list.0 }} - нулевой элемент списка my_list
{{ my_dict.key }} - ключ словаря my_dict "key"
```

Также здесь добавилась ссылка для возврата на главную страницу в начале контента страницы.

Наконец, можно заменить символы `#` в ссылках в шаблоне `index.html` на ссылки на наши статьи:

```
{% for article in articles %}
    ...
    <p><a href="/article/?pk={{ article.pk }}">Подробнее...</a></p>
    ...
{% endfor %}
```

## Создание статей по запросу

После обновления шаблона `article_view.html` сломался предпросмотр статьи по адресу `/articles/add`, т.к. в результирующем шаблоне находятся другие переменные, и контекст из `article_create_view` больше не подходит.

Но теперь сайт позволяет создавать объекты модели `Article` и сохранять их в базу данных, поэтому можно обновить представление `article_create_view` и доработать его до полноценного создания статьи.

Откройте представление **`article_create_view`** из модуля **`webapp/views.py`**. Замените код в ветке для обработки POST-запроса на следующий:

```
title = request.POST.get('title')
text = request.POST.get('content')
author = request.POST.get('author')
article = Article.objects.create(title=title, text=text,
author=author)
context = {
    'article':article
}
return render(request, 'article_view.html', context)
```

Если теперь открыть форму по адресу `http://localhost:8000/articles/add` и добавить новую статью, то она попадёт в базу данных, а вы увидите её на сайте.

Также можно добавить ссылки для навигации между страницами: ссылку на добавление статей на главной странице:

### **index.html**

```
<body>
    <h1>Articles</h1>
    <p><a href="/articles/add/">Добавить</a></p>
    ...
```

И ссылку обратно на главную страницу со страницы добавления:

### **article\_create.html**

```
<body>
    <h1>Create Article</h1>
    <p><a href="/">На главную</a></p>
    ...
```

## Дополнительно

- <https://docs.djangoproject.com/en/4.0/ref/models/fields/> - типы полей в моделях,
- <https://docs.djangoproject.com/en/4.0/ref/contrib/admin/> - настройка админ-панели,
- <https://docs.djangoproject.com/en/4.0/howto/initial-data/> - загрузка начальных данных в БД,
- <https://docs.djangoproject.com/en/4.0/intro/tutorial01/> - официальный tutorial по Django,
- <https://tutorial.djangogirls.org/ru/> - ещё один хороший, годный tutorial по Django на русском.

