Создаём своё первое приложение с Django, часть 2

Продолжаем начатое в первой части учебника. Мы настроим базу данных, создадим свою первую модель и посмотрим на автоматически созданный интерфейс администратора.

Где получить помощь:

При наличии проблем с данной инструкцией, пожалуйста, обратитесь к разделу FAQ Получение помощи.

Настройка базы данных

Теперь откроем файл `mysite/settings.py`. Это обычный модуль Python с набором переменных, которые представляют настройки Django.

По умолчанию в настройках указано использование SQLite. Если вы новичок в базах данных, или просто хотите попробовать Django, это самый простой выбор. SQLite уже включен в Python, и вам не нужно дополнительно что-то устанавливать. Однако, создавая свой первый настоящий проект, вам понадобиться более серьезная база данных, например PostgreSQL.

Если вы хотите использовать другую базу данных, установите необходимые библиотеки и поменяйте следующие ключи в элементе `'default'` настройки `DATABASES`, чтобы они соответсвовали настройкам подключения к вашей базе данных:

- `ENGINE` – один из `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'` или `'django.db.backends.oracle'`. Также доступны дополнительные бэкенды.
- `NAME` – название вашей базы данных. Если вы используете SQLite, база данных будет файлов на вашем компьютере, в этом случае `NAME` содержит абсолютный путь, включая имя, к этому файлу. Значение по умолчанию, `os.path.join(BASE_DIR, 'db.sqlite3')`, создаст файл в каталоге вашего проекта.

Если вы используете не SQLite,вам необходимо указать дополнительно `USER`, `PASSWORD` и `HOST`. Подробности смотрите в описании `DATABASES`.

Для других баз данных, кроме SQLite

Если вы используете не SQLite, убедитесь, что вы создали базу данных. Вы можете сделать это, выполнив «`CREATE DATABASE database_name;`» в консоли вашей базы данных.

Также обратите внимание, что пользователь базы данных из `mysite/settings.py` имеет права создавать базу данных(«create database»). Это позволит автоматически создавать тестовую базу данных.

Если вы используете SQLite, вам ничего не нужно создавать заранее - файл базы данных будет создан автоматически при необходимости.

Отредактируйте `mysite/settings.py` и укажите в `TIME_ZONE` ваш часовой пояс.

Также обратите внимание на настройку `INSTALLED_APPS` в начале файла. Она содержит названия всех приложений Django, которые активированы в вашем проекте. Приложения могут использоваться на разных проектах, вы можете создать пакет, распространить его и позволить другим использовать его на своих проектах.

По умолчанию, `INSTALLED_APPS` содержит следующие приложение, которые предоставляются Django:

- `django.contrib.admin` – интерфейс администратора. Скоро мы его будем использовать.
- `django.contrib.auth` – система авторизации.
- `django.contrib.contenttypes` – фреймверк типов даных.
- `django.contrib.sessions` – фреймверк сессии.
- `django.contrib.messages` – фреймверк сообщений.
- `django.contrib.staticfiles` – фреймверк для работы со статическими файлами.

Эти приложения включены по умолчанию для покрытия основных задач.

Некоторые приложения используют минимум одну таблицу в базе данных, поэтому нам необходимо их создать перед тем, как мы будем их использовать. Для этого выполним следующую команду:

```
$ python manage.py migrate
```


Предыдущий раздел

Создаём своё первое приложение с Django, часть 1

Следующий раздел

Создаём своё первое приложение с Django, часть 3

Эта страница

- Исходный текст

Последнее обновление:

нояб. 23, 2021

Команда `migrate` проверяет настройку `INSTALLED_APPS` и создает все необходимые таблицы в базе данных, указанной в `mysite/settings.py`, применяя миграции, которые находятся в приложении (мы расскажем об этом ниже). Вы увидите сообщение о каждой применённой миграции. Если вам интересно, запустите консоль базы данных и введите `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), `.schema` (SQLite), или `SELECT TABLE_NAME FROM USER_TABLES;` (Oracle), чтобы увидеть таблицы, которые создал Django.

Для минималистов

Как мы сказали выше, приложения по умолчанию включены для большинства случаев, но не все они могут быть необходимы. Если какое-то приложение вам не нужно, закомментируйте или удалите соответствующие строки из `INSTALLED_APPS` перед запуском `migrate`. Команда `migrate` выполнит миграции только для приложений из `INSTALLED_APPS`.

Создание моделей

Теперь создадим ваши модели – по сути структуру вашей базы данных с дополнительными мета-данными.

Философия

Модель — это основной источник данных. Она содержит информацию о наборе полей и о поведении данных, которые вы храните. Django следует принципу DRY. Его цель в определении вашей модели данных в одном месте и автоматической генерации остального на её основе.

Это включает в себя и миграции. В отличии от Ruby On Rails, например, миграции полностью создаются на основе вашего файла моделей и, по своей сути, являются историей того, как Django изменит схему вашей базы данных, чтобы соответствовать текущему состоянию ваших моделей.

В нашем приложении голосования, мы создадим две модели: `Question` и `Choice`. `Question` содержит вопрос и дату публикации. `Choice` содержит два поля: текст ответа и подсчёт голосов. Каждый объект `Choice` связан с объектом `Question`.

Эти понятия представлены в виде классов Python. Отредактируйте файл `polls/models.py`, чтобы он выглядел таким образом:

polls/models.py

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Здесь каждая модель представлена классом, унаследованным от `django.db.models.Model`. Каждая модель содержит несколько атрибутов, каждый из которых представляет поле в таблице базы данных.

Каждое поле представлено экземпляром класса `Field` – например, `CharField` для текстовых полей и `DateTimeField` для полей даты и времени. Это указывает Django какие типы данных хранят эти поля.

Названия каждого экземпляра `Field` (например, `question_text` или `pub_date`) это название поля, в «машинном»(machine-friendly) формате. Вы будете использовать эти названия в коде, а база данных будет использовать их как названия колонок.

Вы можете использовать первый необязательный аргумент конструктора класса `Field`, чтобы определить отображаемое, удобное для восприятия, название поля. Оно используется в некоторых компонентах Django, и полезно для документирования. Если это название не указано, Django будет использовать «машинное» название. В этом примере, мы указали отображаемое название только для поля `Question.pub_date`. Для всех других полей будет использоваться «машинное» название.

Некоторые классы, унаследованные от `Field`, имеют обязательные аргументы. Например, `CharField` требует, чтобы вы передали ему `max_length`. Это используется не только в схеме базы данных, но и при валидации, как мы скоро увидим.

`Field` может принимать различные необязательные аргументы; в нашем примере мы указали `default` значение для votes` равное 0.

Заметим, что связь между моделями определяется с помощью `ForeignKey`. Это указывает Django, что каждый `Choice` связан с одним объектом `Question`. Django поддерживает все основные типы связей: многие-к-одному, многие-ко-многим и один-к-одному.

Активация моделей

Эта небольшая часть кода моделей предоставляет Django большое количество информации, которая позволяет Django:

- Создать структуру базы данных (`CREATE TABLE`) для приложения.
- Создать Python API для доступа к данным объектов `Question` и `Choice`.

Но первым делом мы должны указать нашему проекту, что приложение `polls` установлено.

Философия

Приложения Django «подключаемые»: вы можете использовать приложение в нескольких проектах и вы можете распространять приложение, так как они не связаны с конкретным проектом Django.

Чтобы добавить приложение в проект, необходимо добавить путь до его класса конфигурации в настройку `INSTALLED_APPS`. Класс `PollsConfig` находится в файле `polls/apps.py`, следовательно путь – `'polls.apps.PollsConfig'`. Отредактируйте файл `mysite/settings.py` и добавьте путь в настройку `INSTALLED_APPS`. Это будет выглядеть следующим образом:

mysite/settings.py

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

Теперь Django знает, что необходимо использовать приложение `polls`. Давайте выполним следующую команду:

```
$ python manage.py makemigrations polls
```

Вы увидите приблизительно такое:

```
Migrations for 'polls':
  polls/migrations/0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice
```

Выполняя `makemigrations`, вы говорите Django, что внесли некоторые изменения в ваши модели (в нашем случае мы создали несколько новых) и хотели бы сохранить их в *миграции*.

1

Миграции используются Django для сохранения изменений ваших моделей (и, следовательно, структуры базы данных) — это просто файлы на диске. Если пожелаете, вы можете изучить миграцию для создания ваших моделей, она находится в файле `polls/migrations/0001_initial.py`. Не волнуйтесь, вам не нужно каждый раз их проверять после их генерации, но их формат удобен для чтения на случай, если вы захотите внести изменения самостоятельно.

В Django есть команда, которая выполняет миграции и автоматически обновляет базу данных - она называется `migrate`. Мы скоро к ней вернемся, но сначала давайте посмотрим какой SQL выполнит эта миграция. Команда `sqlmigrate` получает название миграции и возвращает SQL:

```
$ python manage.py sqlmigrate polls 0001
```

Вы увидите приблизительно такое (мы отформатировали результат для читабельности):

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
  ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
```

```
    DEFERRABLE INITIALLY DEFERRED;

COMMIT;
```

Обратите внимание на следующее:

- Полученные запросы зависят от базы данных, которую вы используете. Пример выше получен для PostgreSQL.
- Названия таблиц созданы автоматически из названия приложения(`polls`) и названия модели в нижнем регистре – `question` и `choice`. (Вы можете переопределить это.)
- Первичные ключи (ID) добавлены автоматически. (Вы можете переопределить и это.)
- Django добавляет `"_id"` к названию внешнего ключа. (Да, вы можете переопределить и это.)
- Связь явно определена через `FOREIGN KEY` constraint. Не волнуйтесь о части с `DEFERRABLE`, это просто указание для PostgreSQL не применять ограничения `FOREIGN KEY` до окончания транзакции.
- Учитываются особенности базы данных, которую вы используете. Специфические типы данных такие как `auto_increment` (MySQL), `serial` (PostgreSQL), или `integer primary key` (SQLite) будут использоваться автоматически. Тоже касается и экранирование названий, что позволяет использовать в названии кавычки – например, использование одинарных или двойных кавычек.
- Команда `sqlmigrate` не применяет миграцию к базе данных, вместо этого она выводит SQL запросы на экран, которые Django считает необходимыми для выполнения миграции. Это полезно для проверки того, что Django собирается сделать, или чтобы предоставить вашему администратору базы данных SQL скрипт.

Если необходимо, можете выполнить `python manage.py check`. Эта команда ищет проблемы в вашем проекте не применяя миграции и не изменяя базу данных.

Теперь, выполните команду `migrate` снова, чтобы создать таблицы для этих моделей в базе данных:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

Команда `migrate` выполняет все миграции, которые ещё не выполнялись, (Django следит за всеми миграциями, используя таблицу в базе данных `django_migrations`) и применяет изменения к базе данных, синхронизируя структуру базы данных со структурой ваших моделей.

Миграции — очень мощная штука. Они позволяют изменять ваши модели в процессе развития проекта без необходимости пересоздавать таблицы в базе данных. Их задача изменять базу данных без потери данных. Мы ещё вернемся к ним, а пока запомните эти инструкции по изменению моделей:

- Внесите изменения в модели (в `models.py`).
- Выполните `python manage.py makemigrations` чтобы создать миграцию для ваших изменений
- Выполните `python manage.py migrate` чтобы применить изменения к базе данных.

Две команды необходимы для того, чтобы хранить миграции в системе контроля версий. Они не только помогают вам, но и могут использоваться другими программистами вашего проекта и в продавшее.

О всех возможностях `manage.py` вы можете прочитать в разделе о django-admin.

Поиграемся с API

Теперь, давайте воспользуемся консолью Python и поиграем с API, которое предоставляет Django. Чтобы запустить консоль Python выполните:

```
$ python manage.py shell
```

Мы используем эту команду вместо просто «python», потому что `manage.py` устанавливает переменную окружения `DJANGO_SETTINGS_MODULE`, которая указывает Django путь импорта для файла `mysite/settings.py`.

Теперь, когда вы в консоли, исследуем API базы данных:

```
>>> from polls.models import Choice, Question  # Import the model classes we just wrote.

# No questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())
```

```
# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

Одну минуту. `<Question: Question object (1)>` – крайне непрактичное отображение объекта. Давайте исправим это, отредактировав модель `Question` (в файле `polls/models.py`) и добавив метод `__str__()` для моделей `Question` и `Choice`:

polls/models.py

```
from django.db import models

class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

Важно добавить метод `__str__()` не только для красивого отображения в консоли, но так же и потому, что Django использует строковое представление объекта в интерфейсе администратора.

1

Давайте также добавим свой метод в эту модель:

polls/models.py

```
import datetime

from django.db import models
from django.utils import timezone


class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Мы добавили `import datetime` и `from django.utils import timezone` для использования стандартной библиотеки Python `datetime` и модуля Django для работы с временными зонами `django.utils.timezone` соответственно. Если вы не знакомы, как Python работает с временными зонами, вы можете прочитать об этом в разделе о поддержке временных зон.

Сохраните эти изменения и запустите консоль Python снова, выполнив `python manage.py shell`:

```
>>> from polls.models import Choice, Question

# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
```

```
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

Подробности о работе со связанными объектами смотрите в соответствующем разделе. Подробности об использовании синтаксиса двойного нижнего подчеркивания читайте в разделе о фильтрах полей. Полная информация об API для работы с базой данных содержится в соответствующем разделе.

Введение в интерфейс администратор Django

Философия

Создание интерфейса администратора для добавления, изменения и удаления содержимого сайта – в основном скучная не креативная задача. Django значительно автоматизирует и упрощает эту задачу.

Django создавался для новостных сайтов, у которых есть разделение между публичными страницами и интерфейсом администратора. Менеджеры используют последний для добавления новостей и другого содержимого сайта, это содержимое отображается на сайте. Django позволяет легко создать универсальный интерфейс для редактирования содержимого сайта.

Интерфейс администратора не предназначен для использования пользователями. Он создан для менеджеров и администраторов сайта.

Создание суперпользователя

Первым делом необходимо создать пользователя, который может заходить на интерфейс администратора. Выполните следующую команду:


```
$ python manage.py createsuperuser
```

Введите имя пользователя и нажмите Enter.

```
Username: admin
```

Теперь введите email:

```
Email address: admin@example.com
```

И наконец введите пароль. Вас спросят об этом дважды, чтобы проверить правильность его ввода.

```
Password: **********
Password (again): *********
Superuser created successfully.
```

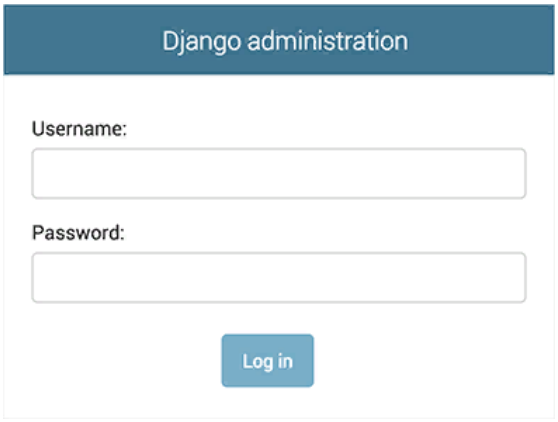
Запускаем сервер для разработки

Интерфейс администратора включен по умолчанию. Давайте запустим встроенный сервер для разработки и посмотрим на него.

Если сервер не запущен, выполните:

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

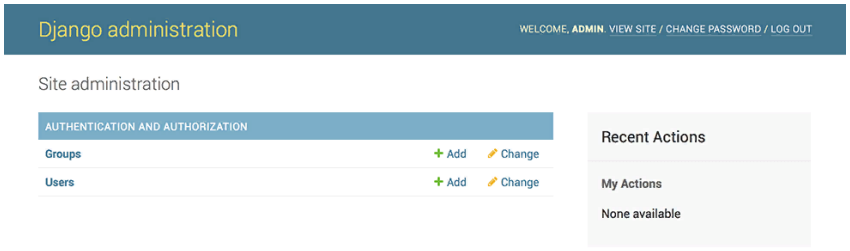
Откроем «/admin/» локального домена в браузере – например, http://127.0.0.1:8000/admin/. Вы должны увидеть страницу авторизации интерфейса администратора:



Т.к. translation включен по умолчанию, страница авторизации может быть на вашем родном языке, зависит от настроек браузера и наличия перевода для вашего языка.

Заходим в интерфейс администратора

Теперь попробуйте войти в админку. Вы должны видеть следующую страницу Django:



Вы должны увидеть несколько разделов: группы и пользователи. Они предоставлены приложением авторизации Django `django.contrib.auth`.

Добавим приложение голосования в интерфейс администратора

А где же наше приложение голосования? Оно не отображается в интерфейсе администратора.

1

Надо сделать ещё одно: нам надо указать, что объекты модели `Question` должны редактироваться в интерфейсе администратора. Для этого создадим файл `polls/admin.py`, и отредактируем следующим образом:

polls/admin.py

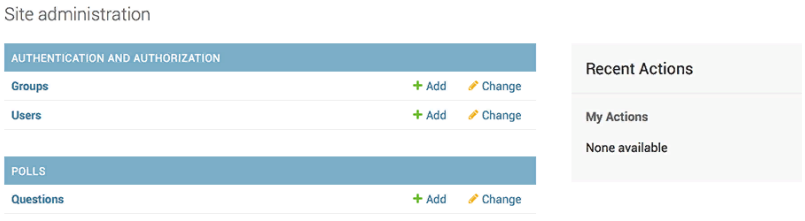
```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

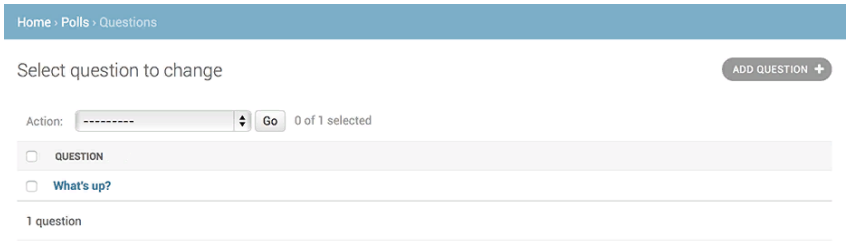
Изучим возможности интерфейса администратора

После регистрации модели `Question` Django отобразит ее на главной странице:

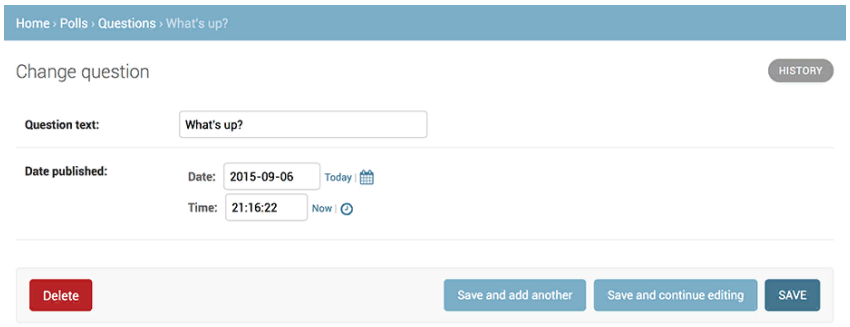


1

Нажмите «Questions». Вы попали на страницу «списка объектов» для голосований. Эта страница содержит все объекты из базы данных и позволяет выбрать один из них для редактирования. Мы видим голосование «What's up?», которое создали в первой части учебника:



Нажмите «What’s up?» чтобы отредактировать его:



Заметим:

- Поля формы формируются на основе описания модели `Question`.
- Для различных типов полей модели (`DateTimeField`, `CharField`) используются соответствующие HTML поля. Каждое поле знает как отобразить себя в интерфейсе администратора.
- К полям `DateTimeField` добавлен JavaScript виджет. Для даты добавлена кнопка «Сегодня» и календарь, для времени добавлена кнопка «Сейчас» и список распространенных значений.

В нижней части страницы мы видим несколько кнопок:

- Save – сохранить изменения и вернуться на страницу списка объектов.
- Save and continue editing – сохранить изменения и снова загрузить страницу редактирования текущего объекта.
- Save and add another – Сохранить изменения и перейти на страницу создания нового объекта.
- Delete – Показывает страницу подтверждения удаления.

Если значение «Date published» не совпадает с временем создания объекта в Части 1 учебника, возможно, вы неверно определили настройку `TIME_ZONE`. Измените ее и перезагрузите страницу.

Измените «Date published», нажав «Today» и «Now». Затем нажмите «Save and continue editing.» Теперь нажмите «History» в правом верхнем углу страницы. Вы увидите все изменения объекта, сделанные через интерфейс администратора, время изменений и пользователя, который их сделал:

Если вы освоили интерфейс администратора, переходите к третьей части этого учебника.