Глава10

Создание платформы

для онлайн-обучения

Из предыдущей главы вы узнали, как добавить на сайт несколько языковых версий, реализовать системы скидочных купонов и рекомендаций товаров.

В этой главе мы начнем с создания нового проекта. Это будет платформа для

онлайн-обучения с собственной системой управления содержимым (Content

Management System, CMS).

Мы рассмотрим следующие темы:

* создание фикстур для моделей;
* наследование моделей;
* реализация собственного поля модели;
* использование обработчиков-классов и примесей;
* работа с наборами форм;
* управление доступом к содержимому сайта с помощью групп и разрешений;

создание CMS.

Создание проекта

В качестве заключительного проекта мы создадим платформу для онлайн-

обучения. В этой главе мы узнаем, как реализовать собственную CMS, которая

позволит создавать курсы и уроки и настраивать их содержимое.

Для начала создайте виртуальное окружение и активируйте его с помощью

этих команд:

mkdir env

virtualenv env/educa

source env/educa/bin/activate

Установите Django с помощью pip:

pip install Django==2.0.5

303

Мы будем работать с изображениями, поэтому нужно установить пакет

Pillow:

pip install Pillow==5.1.0

Теперь создайте новый проект, выполнив команду:

django-admin startproject educa

Перейдите в папку educa и создайте приложения с помощью команд:

cd educa

django-admin startapp courses

Отредактируйте файл settings.py проекта educa и добавьте приложение

courses в список INSTALLEDD\_APPS

INSTALLED\_APPS = [

**'courses.apps.CoursesConfig',**

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Приложение courses активно, пришло время описать модели, которые мы

будем использовать.

Определени моделей для курсов обучения

Платформа для онлайн-обучения работает с такими объектами, как курсы

и предметы.

Каждый курс может быть разбит на несколько модулей.

Каждый модуль может содержать несколько разделов.

Количество модулей и разделов будет настраиваемым.

Каждый раздел содержит различные материалы (Content): тексты, файлы, картинки или видео.

Этот пример демонстрирует, как может быть организован один из курсов:

Subject 1

Course 1

Module 1

Content 1 (image)

Content 2 (text)

Module 2

Content 3 (text)

Content 4 (file)

Content 5 (video)

...

Давайте создадим модели.

**Отредактируйте файл models.py приложения courses** и добавьте в него такой код:

304

from django.db import models

from django.contrib.auth.models import User

class Subject(models.Model): # page 304

title = models.CharField(max\_length=200)

slug = models.SlugField(max\_length=200, unique=True)

class Meta:

ordering = ['title']

def \_\_str\_\_(self):

return self.title

class Course(models.Model): # page 304

owner = models.ForeignKey(User,

related\_name='courses\_created',

on\_delete=models.CASCADE)

subject = models.ForeignKey(Subject,

related\_name='courses',

on\_delete=models.CASCADE)

title = models.CharField(max\_length=200)

slug = models.SlugField(max\_length=200, unique=True)

overview = models.TextField()

created = models.DateTimeField(auto\_now\_add=True)

class Meta:

ordering = ['-created']

def \_\_str\_\_(self):

return self.title

class Module(models.Model): # page 304

course = models.ForeignKey(Course,

related\_name='modules',

on\_delete=models.CASCADE)

title = models.CharField(max\_length=200)

description = models.TextField(blank=True)

def \_\_str\_\_(self):

return self.title

Мы описали модели Subject, Course и Module.

Класс Course содержит следующие поля:

owner– преподаватель, который создал курс;

subjects – предмет, к которому привязан курс. Это внешний Foreign Key ключ-,

на модель Subject;

title – название курса;

slug – слаг курса, мы будем использовать его для формирования челове-

копонятных URLʼов;

overview – текстовое поле типаTextField для создания краткого описания

курса;

created – дата и время создания курса, которые будут устанавливаться

автоматически, т. к. мы указали auto\_now\_add=True.

305

В каждый курс может входить несколько модулей, поэтому для Modul модели

мы определили внешний ключ, Foreign Key, на модель Course.

Откройте консоль и выполните команду создания миграций:

**python manage.py makemigrations**

Вы увидите такой вывод:

Migrations for 'courses':

0001\_initial.py:

- Create model Course

- Create model Module

- Create model Subject

- Add field subject to course

Затем выполните команду:

**python manage.py migrate**

Вы увидите, что применились наша миграция и миграции Django. Вывод бу-

дет содержать такую строку:

Applying courses.0001\_initial... OK

Теперь модели приложения courses синхронизированы с базой данных.

Регистрация моделей на сайте администрирования

Давайте добавим модели курсов на сайт администрирования. **Отредактируйте**

**файл admin.py приложения courses**, добавьте в него такой фрагмент:

from django.contrib import admin

from .models import Subject, Course, Module

@admin.register(Subject)

class SubjectAdmin(admin.ModelAdmin): # page 305

list\_display = ['title', 'slug']

prepopulated\_Melds = {'slug': ('title',)}

class ModuleInline(admin.StackedInline): # page 305

model = Module

@admin.register(Course)

class CourseAdmin(admin.ModelAdmin): # page 305

list\_display = ['title', 'subject', 'created']

list\_filter = ['created', 'subject']

search\_fields = ['title', 'overview']

prepopulated\_fields = {'slug': ('title',)}

inlines = [ModuleInline]

Работа завершена. Мы использовали декоратор@admin.register(), чтобы за-

регистрировать модель на сайте администрирования.

306

Задание начальных данных с помощью фикстур

Иногда необходимо заполнить базу данных какой-то начальной информацией, чтобы объекты были доступны сразу после разворачивания проекта и вам

не пришлось добавлять их в систему вручную. Django предоставляет такую

возможность. Для этого вам понадобятся фикстуры – специальные файлы, которые содержат начальные данные для проекта.

Django поддерживает форматы JSON, XML и YAML. Мы создадим фикстуру,

для того чтобы предварительно заполнить несколько Subject. объектов

Для начала нам понадобится суперпользователь. Создайте его такой командой:

**python manage.py createsuperuser**

Теперь запустите сервер для разработки, выполнив в консоли команду:

**python manage.py runserver**

Откройте в браузере страницу [**http://127.0.0.1:8000/admin/courses/subject/**](http://127.0.0.1:8000/admin/courses/subject/) ,

создайте несколько предметов через сайт администрирования. Список предметов должен будет выглядеть подобным образом:

Рис. 10.1

Список предметов на сайте администрирования

Выполните такую команду:

**python manage.py dumpdata courses --indent=2**

Вы увидите вывод, похожий на этот:

{

"model": "courses.subject",

"pk": 1,

"fields": {

"title": "Mathematics",

"slug": "mathematics"

}Определение моделей для курсов обучения

307

},

{

"model": "courses.subject",

"pk": 2,

"fields": {

"title": "Music",

"slug": "music"

}

},

{

"model": "courses.subject",

"pk": 3,

"fields": {

"title": "Physics",

"slug": "physics"

}

},

{

"model": "courses.subject",

"pk": 4,

"fields": {

"title": "Programming",

"slug": "programming"

}

}

]

У НАС

(mus\_venv) PS E:\Django\_Projects\Music\_training\ Music\_training > python manage.py dumpdata main.course --indent=2

Команда dumpdata по умолчанию выводит информацию по существующим

объектам в консоль в формате JSON. Структура данных, которую мы видим, описывает модели, их поля и значения, так чтобы Django смог загрузить их и сохранить в базу.

Вы можете ограничить список моделей, которые следует выводить. Для этого нужно указать имя приложения и модели в виде app.Model . Также можно применить флаг format, чтобы задать формат вывода. По умолчанию dumpdata выводит результат в консоль. Но это поведение можно изменить с помощью флага output. Флаг indent можно использовать, если вы хотите задать другой размер отступа. Для получения более подробной информации о параметpах dumрdata выполните в консоли

**python manage.py dumpdata -- help**

Сохраните вывод в файл в папку fixture/ приложения orders, выполнив

команды:

**mkdir courses/fixtures**

**python manage.py dumpdata courses --indent=2 output=courses/fixtures/subjects.json**

Запустите сервер для разработки и через сайт администрирования удалите

все предметы, которые создали ранее. Затем загрузите фикстуру в базу данных

с помощью команды:

**python manage.py loaddata subjects.json**

308

Все объекты Subject, которые были добавлены в фикстуру, должны снова по-

явиться в базе данных.

По умолчанию Django ищет файлы фикстуры в папке fixtures / каждого приложения, но можно указать полный путь до них. Также есть возможность задать каталоги, в которых нужно осуществлять поиск, с помощью настройки

FIXTURE\_DIRS.

Фикстуры – это хороший способ заполнить базу данных начальными объектами, а также можно применять их при тестировании.

О том, как использовать фикстуры для тестирования, можно прочесть на

странице https://docs.djangoproject.com/en/2.0/topics/testing/tools/#иxture-loading.

Если вы хотите загружать фикстуры не вручную, а через миграции, обра-

титесь к документации на странице

https://docs.djangoproject.com/en/2.0/topics/migrations/#data-migrations.

**Создание моделей для содержимого курсов**

Наша цель – создать платформу, в которой модули курсов будут содержать раз-

личные типы содержимого. Это может быть текст, картинка, видео или какие-

то файлы. Поэтому нам нужны модели, которые смогут хранить такую инфор-

мацию. В главе 6 вы познакомились с таким понятием, как обобщенные типы.

Они применяются в случаях, когда объекты одной модели могут ссылаться на

несколько объектов других моделей. Мы снова применим тот механизм и соз-

дадим класс Content, объекты которого будут ссылаться на другие модели, представляющие конкретный тип содержимого модуля.

**Отредактируйте файл models.py приложения courses** и импортируйте следую-

щее:

from django.contrib.contenttypes.models import ContentType

from django.contrib.contenttypes.fields import GenericForeignKey

Затем опишите новый класс таким образом:

class Content(models.Model): # page 308

module = models.ForeignKey(Module,

related\_name='contents',

on\_delete=models.CASCADE)

content\_type = models.ForeignKey(ContentType,

on\_delete=models.CASCADE)

object\_id = models.PositiveIntegerField()

item = GenericForeignKey('content\_type', 'object\_id')

Это модель Content. Модуль курса может содержать множество объектов этого

типа, поэтому мы используем Foreign Key на модель Module. Также мы выполнили обобщенную связь, чтобы соединить объекты типа Content с любой другой моделью, представляющей тип содержимого. Помните, чтобы обобщенные связи работали, нам необходимо создать три поля в модели:

309

content\_type – внешний ключ, ForeignKey, на модель ContentType;

object\_id – идентификатор связанного объекта типа PositiveIntegerField ;

item – поле типа GenericForeignKey которое обобщает данные из предыдущих двух.

Только поля content\_type и object\_id будут представлены в базе данных соответствующими столбцами. Полe itеm используется только в Python-коде и позволяет вам получить или задать связанный объект.

Для каждого типа содержимого мы создадим отдельные модели. Они будут

иметь общие данные и отличаться только полем, которое представляет содержимое модуля.

Виды наследования моделей Django

Для реализации таких моделей нам пригодится наследование. В Django существует наследование моделей, которое работает аналогично наследованию класса и может быть реализовано одним из трех способов:

**абстрактные модели** полезны, когда вы хотите описать некоторую об-

щую информацию. Для них не создается таблица в базе данных;

**наследование с помощью нескольких таблиц** применимо, когда каждая

модель в иерархии рассматривается как обособленная и может быть ис-

пользована отдельно от других. В этом случае для каждой модели созда-

ется ее собственная таблица;

**прокси-модели полезны**, когда вы хотите хранить те же данные для всех

моделей, но реализовать для каждой из них какую-то отдельную функ-

циональность (например, создать методы, переопределить или добавить

менеджеры, использовать другие опции класса Meta). Для прокси-моделей таблицы в базе данных не создаются.

Давайте рассмотрим каждый из этих способов подробнее.

Абстрактные модели

Абстрактная модель – это базовый класс. В нем необходимо определить поля,

которые будут общими для всех дочерних классов. Django не добавляет в базу

данных таблицу для абстрактной модели, но для каждого дочернего класса включает столбцы, соответствующие полям родительского класса.

Чтобы указать, что модель является абстрактной, достаточно задать атрибут abstract=True в опциях класса Meta. Django распознает такую настройку и не

создает таблицу в базе данных. Чтобы создать дочернюю модель, просто унаследуйте ее от абстрактной.

Этот пример демонстрирует абстрактную модель Content, и дочернюю, Text:

from django.db import models

class BaseContent(models.Model):

title = models.CharField(max\_length=100)

created = models.DateTimeField(auto\_now\_add=True)

class Meta:

310

abstract = True

class Text(BaseContent):

body = models.TextField()

В этом случае в базе данных будет создана только таблица для Text cомодели

столбцами title, created и body.

Наследование с несколькими таблицами

В случае наследования с несколькими таблицами для каждой модели создается соответствующая таблица. Django делает ссылку типа OneToOneField на родительскую модель из дочерней.

Чтобы применить этот способ, достаточно унаследовать класс дочерней модели от родительской, как это показано в примере:

from django.db import models

class BaseContent(models.Model):

title = models.CharField(max\_length=100)

created = models.DateTimeField(auto\_now\_add=True)

class Text(BaseContent):

body = models.TextField()

Django автоматически добавит поле One To One Field в модельText и создаст две таблицы в базе данных.

Прокси-модели

Они используются,к огда модели хранят одинаковые данныен, о поведение классов отличается (например, у них разные методы или другие опции).

Все они работают с одной таблицей. Чтобы создать прокси-модель, добавьте

в класс Meta атрибут proxy=True.

Давайте посмотрим, как это выглядит, на конкретном примере:

from django.db import models

from django.utils import timezone

class BaseContent(models.Model):

title = models.CharField(max\_length=100)

created = models.DateTimeField(auto\_now\_add=True)

class OrderedContent(BaseContent):

class Meta:

proxy = True

ordering = ['created']

def created\_delta(self):

return timezone.now() - self.created

Здесь мы создали прокси-модель OrderedContent. Она добавляет сортировку

по умолчанию для QuerySetʼов и методa created\_delta(). Обе модели работают

с одной и той же таблицей в базе данных, и обратиться к объектам можно через ORM для каждой модели.

311

**Создание моделей для содержимого курса**

Модель Content приложения courses определяет обобщенную связь с различными типами содержимого. Мы создадим несколько моделей для каждого типа.

Все они будут иметь общие и некоторые специфичные поля. Мы будем использовать абстрактную модель.

**Отредактируйте файл models.py приложения courses** и добавьте в него такой

код:

**class ItemBase(models.Model):** # page 311

**owner = models.ForeignKey(User,**

**related\_name='%(class)s\_related',**

**on\_delete=models.CASCADE)**

**title = models.CharField(max\_length=250)**

**created = models.DateTimeField(auto\_now\_add=True)**

**updated = models.DateTimeField(auto\_now=True)**

**class Meta:**

**abstract = True**

**def \_\_str\_\_(self):**

**return self.title**

**class Text(ItemBase):** # page 311

**content = models.TextField()**

**class File(ItemBase):** # page 311

**file = models.FileField(upload\_to='files')**

**class Image(ItemBase):** # page 311

**file = models.FileField(upload\_to='images')**

**# может быть не file а image ?**

**class Video(ItemBase):** # page 311

**url = models.URLField()**

В этом фрагменте вы создали абстрактную модeль ItеmBase, задав в опциях

класса Meta атрибут abstract=True. Она содержит четыре пoля: оwner, title , created и updated, – которые будут общими для всех дочерних моделей. owner Поле содержит данные пользователя, который создал объект.

Так как в дочерних классах будет присутствовать это поле, необходимо задать related\_name для каждого из них. Django задает связанное наименовaние relаted\_name в виде % (class)s, но мы определили его как '% ( class)s\_related' . Таким образом, объекты каждой дочерней модели будут доступны по именам tеxt\_related, file\_related , image\_related и video\_related.

Мы определили несколько типов содержимого, унаследованных от класса

ItemBase:

* Text – для текста;
* File – для файлов, например PDF;
* Image – для картинок;
* Video – для видео. Мы применили поле URL Field, чтобы сохранять URL видео для его скачивания.

312

Каждая дочерняя модель определяет дополнительные поля. Таблицы в базе

данных будут созданы для моделeй, Tеxt File , Image и Video. Для ItemBaseDjango не добавит таблицу, т. к. эта модель является абстрактной.

Отредактируйте **модель Content**, которую создали раньше, измените ее поле,

**content\_type**, как показано ниже:

**content\_type = models.ForeignKey(ContentType,** # page 312

**on\_delete=models.CASCADE,**

**limit\_choices\_to={'model\_\_in':(**

**'text',**

**'video',**

**'image',**

**'file')})**

Мы добавили атрибут limit\_choices\_to, чтобы ограничить типы содержимого ContentType, которые могут участвовать в связи. Чтобы фильтровать объекты

ContentType при запросах, указали условие model\_\_in и значения 'text' , 'video' ,

'image' и 'file' .

Давайте создадим миграции. Выполните команду:

**python manage.py makemigrations**

Вы увидите такой вывод:

Migrations for 'courses':

courses/migrations/0002\_content\_file\_image\_text\_video.py

- Create model Content

- Create model File

- Create model Image

- Create model Text

- Create model Video

Запустите синхронизацию моделей с базой данных:

**python manage.py migrate**

Вывод в консоли будет содержать такую строку:

Applying courses.0002\_content\_file\_image\_text\_video... OK

Мы создали модели, которые позволят сохранять различное содержимое для

модулей курсов, но упустили небольшую деталь. Модули в курсах и содержимое в модулях должны следовать в определенном порядке. Поэтому нам понадобится поле, в котором он будет соблюдаться.

**Создание собственных типов полей для модели**

В Django предопределено множество полей, с помощью которых можно созда-

вать модели. Но вы можете создать собственное поле, например для того, чтобы хранить особый тип данных или изменить поведение стандартных классов.

Нам нужно поле, которое позволит определить порядок для содержимого

курсов. Самый простой способ – использовать поле PositiveIntegerField. Целочисленные значения будут задавать порядок объектов. Но мы не ограничимся

313

этим, а создадим на основе класса PositiveIntegerField свое поле, которое будет

выполнять некоторые дополнительные действия.

В нашем поле будут реализованы две дополнительные функции:

* автоматическое назначение порядкового номера, если он не был задан

явно. Когда создается новый объект и пользователь не указывает порядок, поле будет заполняться автоматически, основываясь на том, сколько

объектов уже создано для модуля. Например, если уже есть два объекта

с порядковыми номерам 1 или 2, то новому будет присвоен 3;

* сортировка объектов по порядку номеров. Модули курсов и содержимое

модулей всегда будут возвращаться отсортированными внутри своего

родительского объекта.

**Создайте файл fields.py в папке приложения courses** и добавьте в него такой

код:

from django.db import models

from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField): # page 313

def \_\_init\_\_(self, for\_fields=None, \*args, \*\*kwargs):

self.for\_fields = for\_fields

super(OrderField, self).\_\_init\_\_(\*args, \*\*kwargs)

def pre\_save(self, model\_instance, add): # page 313

if getattr(model\_instance, self.attname) is None:

# Значение пусто.

try:

qs = self.model.objects.all()

if self.for\_fields:

# Фильтруем объекты с такими же значениями полей,

# перечисленных в "for\_fields".

query = {field: getattr(model\_instance, field)\

for field in self.for\_fields}

qs = qs.filter(\*\*query)

# Получаем заказ последнего объекта.

last\_item = qs.latest(self.attname)

value = last\_item.order + 1

except ObjectDoesNotExist:

value = 0

setattr(model\_instance, self.attname, value)

return value

else:

return super(OrderField, self).pre\_save(model\_instance, add)

Это класс поля OrderField. Он наследуется от PоsitiveIntegerField, который

определен в Django. Конструктор принимает необязательный параметр for\_fields , чтобы определить поле родительского объекта, относительно которого будет вычислен порядок.

Мы переопределили методe pre\_save() класса PositiveIntegerField. Он выполняется перед тем, как Django сохранит поле в базу данных. В этом методе мы выполняем следующие действия:

314

1) проверяем, существует ли такое значение для объектов модели. Для того

чтобы получить имя поля, по которому оно было определено в модели,

обращаемся к атрибуту self.attname. Если значение поля равнo Nоne, рассчитываем, чему оно должно быть равно:

- формируем QuerySet, чтобы получить все объекты модели. Класс модели, для которой определено текущее поле, получаем через запись self.model;

- если был задан параметр for\_fields , получаем для текущего объекта

значения этих полей и фильтруем QuerySet по ним. Так мы получим

только те объекты, которые принадлежат одному родительскому, например все модули курса;

- получаем объект с максимальным значением порядкового номера из

результата фильтрации с помощью записи last\_ item = qs.latest(self.attname.)

Если не найдено ни одного объекта, присваиваем текущему порядковый номер 0;

- если объект найден, то присваиваем текущему номер, больший на

единицу;

2) если поле заполнено пользователем, ничего не делаем.

Когда вы создаете собственные поля для моделей, старайтесь делать их универсальными. Избегайте явного задания данных или полей моделей, от которых они могут зависеть. Ваши поля должны быть применимы не для одной модели, а для всех.

Больше информации о создании своих классов для полей моделей мож-

но найти на странице https://docs.djangoproject.com/en/2.0/howto/custom-model-fields/.

**Добавление собственного поля сортировки в модели**

Давайте применим в модели поле, которое мы только что создали.

Отредактируйте **файл models.py** **приложения courses,** импортируйте класс OrderField и добавьте поле в модель Module:

from .fields import OrderField

class Module(models.Model):

# ...

order = OrderField(blank=True, for\_fields=['course'])# page 314поле сортировки в модели

Новое поле называется order. Оно будет рассчитываться автоматически для

каждого модуля в рамках одного курса, т. к. мы указали for \_fields=['course'] .

Таким образом, при создании нового модуля его порядок будет больше на единицу, чем у предыдущего модуля курса.

**Теперь отредактируйте\_\_str\_\_() метод модели Module,** как показано ниже:

**class Module(models.Model):**

**# ...**

**def \_\_str\_\_(self):** # page 314

**return '{}. {}'.format(self.order, self.title)**

315

Содержимое модулей также должно сортироваться по порядку.

**Добавьте новое поле OrderField в модель Content:**

**class Content(models.Model):**

**# ...**

**order = OrderField(blank=True, for\_fields=['module'])** # page 315

Здесь мы использовали запись for\_ fields=['module'], следовательно, порядковые номера объектов будут задаваться в рамках одного модуля. **Теперь определите сортировку по умолчанию в классе Meta для моделей Module и Content:**

**class Module(models.Model):** # page 315

**# ...**

**class Meta:**

**ordering = ['order']**

**class Content(models.Model):** # page 315

**# ...**

**class Meta:**

**ordering = ['order']**

На текущий момент модели Module и Content должны выглядеть таким образом:

**class Module(models.Model):** # page 315

**course = models.ForeignKey(Course,**

**related\_name='modules',**

**on\_delete=models.CASCADE)**

**title = models.CharField(max\_length=200)**

**description = models.TextField(blank=True)**

**order = OrderField(blank=True, for\_fields=['course'])**

**class Meta:**

**ordering = ['order']**

**def \_\_str\_\_(self):**

**return '{}. {}'.format(self.order, self.title)**

**class Content(models.Model):** # page 315

**module = models.ForeignKey(Module,**

**related\_name='contents',**

**on\_delete=models.CASCADE)**

**content\_type = models.ForeignKey(ContentType,**

**on\_delete=models.CASCADE,**

**limit\_choices\_to={'model\_\_in':(**

**'text',**

**'video',**

**'image',**

**'file')})**

**object\_id = models.PositiveIntegerField()**

**item = GenericForeignKey('content\_type', 'object\_id')**

**order = OrderField(blank=True, for\_fields=['module'])**

**class Meta:**

**ordering = ['order']**

316

Давайте создадим миграцию, которая перенесет эти изменения в базу данных.

Откройте консоль и выполните команду:

**python manage.py makemigrations courses**

Вы увидите вывод, который содержит такие строки:

You are trying to add a non-nullable field 'order' to content without a default; we can't do that (the database needs something to populate existing rows).

Please select a fix:

1) Provide a one-off default now (will be set on all existing rows with a null value for

this column)

2) Quit, and let me add a default in models.py Select an option:

Django сообщает о том, что мы не указали значение по умолчанию для полей

order. Если поле определено с параметром null=True, то для всех существующих

записей будет проставлено пустое значение, а Django добавит миграцию авто-

матически, без запроса действия. Мы можем задать значение по умолчанию

или отменить создание миграции и добавить параметр default в поля order мо-

делей, после чего снова выполнить команду.

**Введите1 и нажмите Enter**, чтобы проставить значение по умолчанию для

существующих объектов. Вы увидите такой вывод:

Please enter the default value now, as valid Python

The datetime and django.utils.timezone modules are available, so you can do

e.g. timezone.now

Type 'exit' to exit this prompt

>>>

**Введите 0 и снова нажмите Enter.** Django попросит вас ввести значение по

умолчанию для модели Module. Выберите первую **опцию и задайте 0.** Наконец, миграция будет создана, вы увидите такой вывод:

Migrations for 'courses':

courses/migrations/0003\_auto\_20180326\_0704.py

- Change Meta options on content

- Change Meta options on module

- Add field order to content

- Add field order to module

Примените изменения в базе данных, выполнив команду:

**python manage.py migrate**

Вы увидите сообщение о том, что миграции успешно применились, оканчи-

вающееся такой строкой:

Applying courses.0003\_auto\_20180326\_0704... OK

Сейчас нужно протестировать наше поле! Откройте консоль и выполните

команду:

**python manage.py shell**

317

Создайте новый курс:

>>> from django.contrib.auth.models import User

>>> from courses.models import Subject, Course, Module

>>> user = User.objects.last()

>>> subject = Subject.objects.last()

>>> c1 = Course.objects.create(subject=subject, owner=user, title='Course 1', slug='course1')

Курс создан и сохранен в базу данных. Добавим в него модулей и посмотрим,

какие порядковые номера будут им присвоены. Это можно выполнить следующим образом:

>>> m1 = Module.objects.create(course=c1, title='Module 1')

>>> m1.order

0

Класс OrderField задал значение по умолчанию, 0, т.к. это первый объект модели Module для соответствующего курса. Теперь создайте второй модуль:

>>> m2 = Module.objects.create(course=c1, title='Module 2')

>>> m2.order

1

Порядковый номер был вычислен автоматически путем добавления единицы к номеру последнего модуля этого курса. Давайте добавим третий модуль,

явно указав его номер:

>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)

>>> m3.order

5

Вы можете убедиться, что если задать номер явно, то класс OrderFiled не будет выполнять никаких действий и просто сохранит переданное число.

Добавьте четвертый модуль:

>>> m4 = Module.objects.create(course=c1, title='Module 4')

>>> m4.order

6

Мы не указали порядок, поэтому он был задан автоматически. То, как мы

реализовали поле OrderField, не гарантирует, что порядковые номера будут по-

следовательными. Но при автоматическом формировании поле никогда не за-

даст предыдущее значение, а использует следующий порядковый номер.

Давайте посмотрим, что произойдет, если добавить новый курc и один модуль для него:

>>> c2 = Course.objects.create(subject=subject, title='Course 2',

slug='course2', owner=user)

>>> m5 = Module.objects.create(course=c2, title='Module 1')

>>> m5.order

0

318

Чтобы вычислить номер для нового модуля, использовались только модули

второго курса. Так как это первый объект, порядковый номер задан 0. Это как

происходит потому, что при определении поля order модели Module мы указали атрибут for\_felds=['course'] .

Поздравляем! Вы только что создали собственное поле для модели.

**Создание системы управления содержимым (CMS)**

В предыдущих разделах мы описали модели для платформы онлайн-обучения.

Теперь необходимо создать систему управления содержимым сайтов (CMS).

CMS позволит преподавателям создавать курсы и управлять их модулями. Нам

нужно реализовать такие возможности, как:

* авторизация в системе;
* отображение списка курсов преподавателя;
* создание, редактирование и удаление курсов;
* добавление модулей к курсам;
* добавление содержимого различных типов в модули.

Добавление системы аутентификации

Мы будем использовать стандартную подсистему аутентификации Django.

Преподаватели и студенты курсов будут представлены моделью User фреймворка. Таким образом, за доступ пользователя к аккаунту будет отвечать пакет django.contrib.auth.

**Отредактируйте файл urls .py проекта music\_training** и добавьте обработчики входа и выхода подсистемы аутентификации Django:

from django.contrib import admin

from django.urls import path

from django.contrib.auth import views as auth\_views

urlpatterns = [

path('accounts/login/', auth\_views.LoginView.as\_view(), name='login'), # page 318 обработчик входа подсистемы аутентификации

path('accounts/logout/', auth\_views.LogoutView.as\_view(http\_method\_names = ['get', 'post', 'options']), name='logout'), # page 318 обработчик выхода подсистемы аутентификации

path('admin/', admin.site.urls), # page 318

]

Создание шаблонов аутентификации

**Добавьте папку для шаблонов в каталог приложения courses и создайте в ней**

**такую структуру файлов:**

templates/

base.html

registration/

login.html

logged\_out.html

319

Перед тем как определить шаблоны страниц сайта, нам нужно определить

базовый шаблон.

**Отредактируйте файл base.html** и добавьте в него такой код:

{% load staticfiles %}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>{% block title %}Music\_training {% endblock %}</title>

<link href="{% static "css/base.css" %}" rel="stylesheet">

</head>

<body>

<div id="header">

<a href="/" class="logo">Music\_training </a>

<ul class="menu">

{% if request.user.is\_authenticated %}

<li><a href="{% url "logout" %}">Sign out</a></li>

{% else %}

<li><a href="{% url "login" %}">Sign in</a></li>

{% endif %}

</ul>

</div>

<div id="content">

{% block content %}

{% endblock %}

</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/

3.3.1/jquery.min.js"></script>

<script>

$(document).ready(function() {

{% block domready %}

{% endblock %}

});

</script>

</body>

</html>

Это базовый шаблон для всех страниц. Он содержит три блока:

* title – блок для подстановки заголовка;
* content – блок для содержимого страницы;
* domready – блок для JavaScript-кода, размещенный внутри функции $document.ready(). Он позволит задать операции, которые будут выполнены после загрузки DOM-страницы.

**CSS-стили находятся в папке static/ каталога приложения courses в коде –**

**примере к этой главе.**

**Скопируйте эту папку в свой проект по тому же пути, чтобы они применились.**

**Отредактируйте файл registration/login.html** и добавьте в него такой фрагмент:

320

{% extends "base.html" %}

{% block title %}Log-in{% endblock %}

{% block content %}

<h1>Log-in</h1>

<div class="module">

{% if form.errors %}

<p>Your username and password didn't match. Please try again.</p>

{% else %}

<p>Please, use the following form to log-in:</p>

{% endif %}

<div class="login-form">

<form action="{% url 'login' %}" method="post">

{{ form.as\_p }}

{% csrf\_token %}

<input type="hidden" name="next" value="{{ next }}" />

<p><input type="submit" value="Log-in"></p>

</form>

</div>

</div>

{% endblock %}

Это стандартный шаблон для страницы входа Django.

**Отредактируйте файл registration/logged\_out.html,** добавив такой код:

{% extends "base.html" %}

{% block title %}Logged out{% endblock %}

{% block content %}

<h1>Logged out</h1>

<div class="module">

<p>You have been successfully logged out.

You can <a href="{% url "login" %}">log-in again</a>.</p>

</div>

{% endblock %}

Эта страница будет показана пользователю, после того как он выйдет из

своего аккаунта.

В courses/view приложения добавить:

class LogoutView(View): # нашел что-то похожее  
 def get(self, request):  
 logout(request)  
 return redirect('login')

Запустите сервер для разработки командой:

**python manage.py runserver**

**Откройте в браузере http://127.0.0.1:8000/accounts/login/**. Вы увидите такую

страницу входа:

321

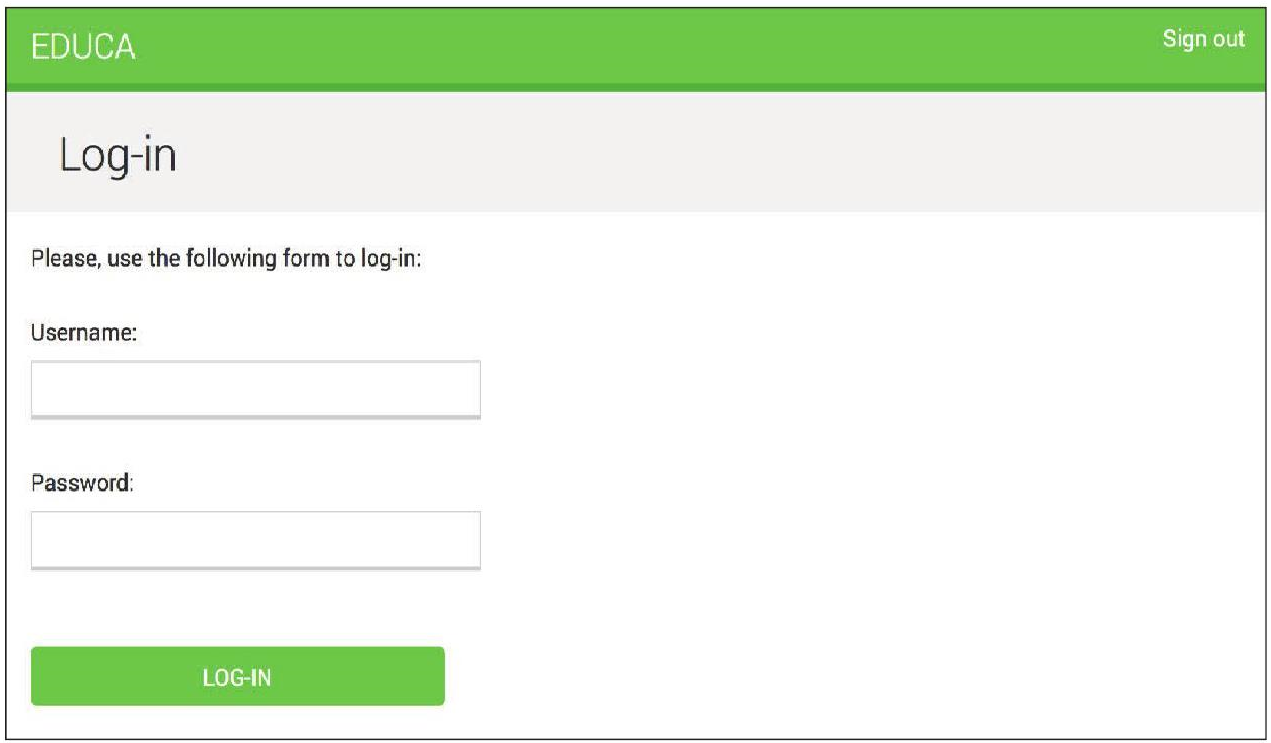


Рис. 10.2

Страница входа в аккаунт

**Определение обработчиков-классов**

Мы определим обработчики для создания, редактирования и удаления курсов,

используя для этого не функции, а классы.

**Откройте views.py файл приложения courses и вставьте в него такой фрагмент:**

from django.views.generic.list import ListView

from .models import Course

class ManageCourseListView(ListView): # page 321

model = Course

template\_name = 'courses/manage/course/list.html'

def get\_queryset(self): # page 321

qs = super(ManageCourseListView, self).get\_queryset()

return qs.filter(owner=self.request.user)

Это класс ManageCourseListView, который будет выступать в роли обработчика

запросов. Он наследуется от класса ListView Django. Мы переопределили метод get\_queryset(), чтобы получать курсы, созданные текущим пользователем.

Чтобы не дать пользователям возможности редактировать курсы, которые они

не создавали, мы переопределим этот метод аналогичным образом в соответствующих обработчиках. Когда появляется необходимость в реализации одинаковой логики в нескольких обработчиках, на помощь приходят примеси.

Использование примесей для обработчиков

миксины

Примесь, или миксин, – это класс, который используется при множественном

наследовании. При определении вашего класса можно задействовать несколько примесей, каждая из которых добавит часть функций в класс. Примеси удобны в двух случаях:

322

* вы хотите использовать несколько различных функций в рамках вашего

класса;

* вы хотите реализовать одну и ту же функциональность в нескольких

классах.

Django предоставляет набор примесей для обработчиков на основе классов.

Более подробно о них можно прочесть на странице https://docs.djangoproject.com/en/2.0/topics/class-based-views/mixins/.

Мы создадим собственную примесь, чтобы не дублировать код для обработчиков действий с курсами.

**Откройте файл views.py приложения courses** и измените его таким образом:

**Music\_training/courses/views.py**

from django.urls import reverse\_lazy

from django.views.generic.list import ListView

from django.views.generic.edit import CreateView, UpdateView, DeleteView

from .models import Course

class OwnerMixin(object): # page 322

def get\_queryset(self):

qs = super(OwnerMixin, self).get\_queryset()

return qs.filter(owner=self.request.user)

class OwnerEditMixin(object): # page 322

def form\_valid(self, form):

form.instance.owner = self.request.user

return super(OwnerEditMixin, self).form\_valid(form)

class OwnerCourseMixin(OwnerMixin): # page 322

model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin): # page 322

fields = ['subject', 'title', 'slug', 'overview']

success\_url = reverse\_lazy('manage\_course\_list')

template\_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView): # page 322

template\_name = 'courses/manage/course/list.html' # page 322

class CourseCreateView(OwnerCourseEditMixin, CreateView): # page 322

pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView): # page 322

pass

class CourseDeleteView(OwnerCourseMixin, DeleteView): # page 322

template\_name = 'courses/manage/course/delete.html' # page 322

success\_url = reverse\_lazy('manage\_course\_list')

В этом фрагменте мы определили две примеси: OwnerMixin и OwnerEditMixin.

Они будут добавлены к нашим обработчикам вместе с такими классами Django,

как ListView, CreateView, UpdateView и DeleteView. Примесь OwnerMixin определяет метод get\_queryset(). Он используется для получения базового QuerySetʼа, с которым будет работать обработчик. Мы переопределили этот метод, так чтобы получать только объекты, владельцем которых является текущий пользователь (request.user).

323

Примесь OwnerEditMixin определяет метод form\_valid(). Django вызывает его

для обработчиков, которые наследуются Model от FormMixin и работают с формами и модельными формами, например CreateView или UpdateView. Методы выполняются, когда форма успешно проходит валидацию. Поведение по умолчанию для примеси Django – сохранение объекта в базу данных (для модельных форм) и перенаправление пользователя на страницу по адресу success\_url (для обычных форм). Мы переопределили этот метод, чтобы автоматически заполнять поле owner сохраняемого объекта.

Примесь OwnerMixin можно применять для любого обработчика, который работает с моделью, содержащей owner поле.

Мы также создали класс OwnerCourseMixin, который наследуется от OwnerMixin, и добавили для него атрибут model – модель, с которой работает обработчик.

Еще одна примесь, OwnerCourseEditMixin, содержит такие атрибуты:

* fields – поля модели, из которых будет формироваться объект обработчиками CreateView и UpdateView;
* success\_url – адрес, на который пользователь будет перенаправлен после

успешной обработки формы классами CreateView и UpdateView или их наследниками. Мы указали шаблон URLʼа с именем manage\_course\_list, который добавим чуть позже.

Наконец, мы создали такие обработчики-наследники от класса OwnerCourseMixin, как:

* ManageCourseListView – список курсов, созданных пользователем. Наследуется от OwnerCourseMixin и ListView;
* CourseCreateView – использует модельную форму для создания нового курса. При создании объекта из данных запроса учитывает поля, определенные в родительском классе OwnerCourseEditMixin. Является наследником CreateView;
* CourseUpdateView –позволяет владельцу курса редактировать его. Наследник двух классов – OwnerCourseEditMixin и UpdateView;
* CourseDeleteView – наследуется от примеси OwnerCourseMixin и обработчика DeleteView. Задает атрибут success\_url – адрес, на который пользователь будет перенаправлен после успешного удаления объекта.

Работа с группами и правами

Мы создали классы для обработки действий с курсами. Сейчас любой пользователь может получить доступ к этим обработчикам, зная URL.

Необходимо исключить такую возможность и разрешить редактирование курсов только их владельцам. В подсистему аутентификации Django включена система разрешений, которая позволяет администратору создать группы пользователей и ограничения на доступ к тем или иным действиям. Это то, что нам нужно.

Давайте создадим группу «Владелец курса» (Instructors) и разрешим ей доступ

к обработчикам создания, изменения и удаления курсов.

**Запустите сервер разработки и перейдите по адресу**

**http://127.0.0.1:8000/admin/auth/group/add/,** чтобы создать новую группу пользователей.

324

Добавьте название Instructors и выберите все разрешения приложения courses, кроме разрешений на модель Subject, как показано ниже:

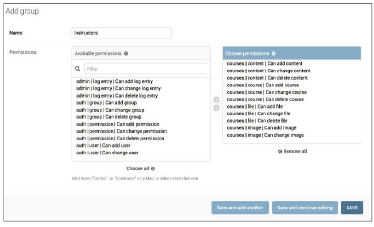


Рис. 10.3

Создание группы пользователей «Владелец курса»

Как вы можете заметить, для каждой модели существует три разрешения:

can add, can change и can delete. После выбора сохраните группу, нажав кнопку

SAVE.

Django создает разрешения для моделей проекта автоматически, но можно

реализовать собственные разрешения. Более подробно об этом можно прочесть на странице https://docs.djangoproject.com/en/2.0/topics/auth/customizing/

#custom-permissions.

Откройте страницу <http://127.0.0.1:8000/admin/auth/user/add/> и создайте нового пользователя. Добавьте его в группу «Владелец курса», как показано ниже:



Рис. 10.4

Добавление пользователя в группу «Владелец курса»

325

Разрешения группы распространяются на всех пользователей, состоящих

в ней. Но Django позволяет выдавать их и индивидуально, т. е. для каждого пользователя. Это можно сделать и через сайт администрирования. У пользователя есть атрибут is\_superuser, который определяет его принадлежность к суперпользователям сайта. Если флаг равен True, то система разрешений Django даст доступ такому пользователю ко всем возможностям системы, несмотря на его связи с группами.

**Ограничение доступа к обработчикам-классам**

Мы хотим ограничить доступ к обработчикам, так чтобы только пользователи,

имеющие права на добавление, редактирование или удаление объектов Course, могли к ним обратиться. Для этого применим две примеси, которые определены в пакете django.contrib.auth:

* LoginRequiredMixin – примесь разрешает доступ к обработчику только авторизованным пользователям, аналогична декоратoру lоgin\_required для обработчиков-функций;
* PermissionRequiredMixin – разрешает доступ только пользователям, у которых есть заданное разрешение. Исключением являются суперпользователи. Для них обработчик будет доступен всегда.

**Отредактируйте файл views.py приложения courses** и импортируйте следующее:

**from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin**

Добавьте еще один родительский класс LoginRequiredMixin в примесь OwnerCourseMixin:

class OwnerCourseMixin(OwnerMixin, LoginRequiredMixin): # 325

model = Course

fields = ['subject', 'title', 'slug', 'overview']

success\_url = reverse\_lazy('manage\_course\_list')

Теперь добавьте атрибут permission\_required, чтобы задать разрешения, которые будут проверяться при доступе к обработчикам:

class CourseCreateView ( # # 325

PermissionRequiredMixin,

OwnerCourseEditMixin,

CreateView):

permission\_required = 'courses.add\_course'

class CourseUpdateView (PermissionRequiredMixin, # 325

OwnerCourseEditMixin,

UpdateView):

permission\_required = 'courses.change\_course'

class CourseDeleteView (PermissionRequiredMixin, # 325

OwnerCourseMixin,

DeleteView):

template\_name = 'courses/manage/course/delete.html'

success\_url = reverse\_lazy('manage\_course\_list')

permission\_required = 'courses.delete\_course'

326

Примесь PermissionRequiredMixin добавляет проверку наличия у пользователя

разрешения, указанного в атрибуте permission\_required. Теперь наши обработчики доступны только пользователям, которые имеют соответствующие разрешения.

Давайте добавим шаблоны URLʼов.

**Создайте новый urls.py файл в папке приложения courses** и вставьте в него такой код:

from django.urls import path

from . import views

urlpatterns = [

path('mine/', # 326

views.ManageCourseListView.as\_view(),

name='manage\_course\_list'),

path('create/', # 326

views.CourseCreateView.as\_view(),

name='course\_create'),

path('<pk>/edit/', # 326

views.CourseUpdateView.as\_view(),

name='course\_edit'),

path('<pk>/delete/', # 326

views.CourseDeleteView.as\_view(),

name='course\_delete'),

]

Это шаблоны URLʼов, по которым будет формироваться адрес до обработчи-

ков создания, редактирования и удаления курсов.

**Отредактируйте urls.py файл** **проекта** educa и подключите шаблоны, как показано ниже:

from django.urls import path, include

urlpatterns = [

path('accounts/login/', auth\_views.LoginView.as\_view(), name='login'),

path('accounts/logout/', auth\_views.LogoutView.as\_view(http\_method\_names = ['get', 'post', 'options']), name='logout'),

path('admin/', admin.site.urls),

path('course/', include('courses.urls')),

]

Самое время добавить HTML-шаблоны для обработчиков. **Создайте в ката-**

**логе templates/ приложения courses такую структуру папок и файлов:**

courses/

manage/

course/

list.html

form.html

delete.html

**Отредактируйте файл courses/manage/course/list.html**, добавьте в него сле-

дующий код:

327

{% extends "base.html" %}

{% block title %}My courses{% endblock %}

{% block content %}

<h1>My courses</h1>

<div class="module">

{% for course in object\_list %}

<div class="course-info">

<h3>{{ course.title }}</h3>

<p>

<a href="{% url "courses:course\_edit" course.id %}">Edit</a>

<a href="{% url " courses:course\_delete" course.id %}">Delete</a>

</p>

</div>

{% empty %}

<p>You haven't created any courses yet.</p>

{% endfor %}

<p>

<a href=' courses:course\_create' class="button">Create new course</a>

</p>

</div>

{% endblock %}

Этот шаблон будет использоваться обработчиком Manage CourseListView. Он

отображает список курсов, созданных текущим пользователем. Мы добавили

ссылку на страницу редактирования курса и на страницу создания нового.

**Запустите сервер разработки командой python manage.py runserver** и

перейдите по адресу

<http://127.0.0.1:8000/accounts/login/?next=/course/mine/> ,

зарегистрируйтесь как пользователь с группой «Владелец курса».

После входа вы будете перенаправлены на http://127.0.0.1:8000/course/mine/ и увидите такую страницу:

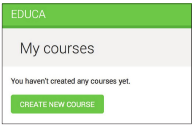


Рис. 10.5 Список курсов,созданных пользователем

328

На этой странице будут появляться курсы, владельцем которых является

пользователь.

Давайте добавим шаблон, который покажет форму для создания и редактирования курсов.

**Откройте файл courses/manage/course/form.html** ивставьте в него такой код:

{% extends "base.html" %}

{% block title %}

{% if object %}

Edit course "{{ object.title }}"

{% else %}

Create a new course

{% endif %}

{% endblock %}

{% block content %}

<h1>

{% if object %}

Edit course "{{ object.title }}"

{% else %}

Create a new course

{% endif %}

</h1>

<div class="module">

<h2>Course info</h2>

<form action="." method="post">

{{ form.as\_p }}

{% csrf\_token %}

<p><input type="submit" value="Save course"></p>

</form>

</div>

{% endblock %}

Шаблон form.html используется двумя обработчиками: CourseCreateView и CourseUpdateView. В нем мы проверяем, есть ли переменная object в контексте. Если есть, показываем страницу редактирования курса, в противном случае – создания.

**Откройте в браузере** [**http://127.0.0.1:8000/course/mine**/](http://127.0.0.1:8000/course/mine/) и нажмите на кнопку

CREATE NEW COURSE.

Вы увидите такую страницу:

329

Рис. 10.6 Форма создания курса

Заполните форму и нажмите на кнопку SAVE COURSE. Курс сохранится

в базу данных, а вы будете перенаправлены на список курсов:

Рис. 10.7 Список курсов содержит добавленный объект

330

Кликните по ссылке Edit рядом с курсом, который вы только что создали. Вы

увидите ту же форму, но поля будут заполнены данными курса.

Наконец, **отредактируйте файл courses/manage/course/delete.html** и вставьте

в него такой фрагмент:

{% extends "base.html" %}

{% block title %}Delete course{% endblock %}

{% block content %}

<h1>Delete course "{{ object.title }}"</h1>

<div class="module">

<form action="" method="post">

{% csrf\_token %}

<p>Are you sure you want to delete "{{ object }}"?</p>

<input type="submit" class"button" value="Con᣿rm">

</form>

</div>

{% endblock %}

Эта страница используется обработчиком CourseDeleteView. Он унаследован

от стандартного класса DjangoDeleteView, и реализует удаление объекта.

Откройте браузер и нажмите на ссылку Delete рядом с курсом. Вы увидите

страницу подтверждения удаления:

Рис. 10.8 Подтверждение действия обработчика CourseDeleteView

Нажмите кнопку CONFIRM. Курс будет удален, а вы перенаправитесь на

страницу списка курсов.

На текущий момент владельцы курсов могут управлять ими: создавать, редактировать и удалять. Следующий шаг – добавление CMS, с помощью которой можно будет управлять модулями и их содержимым.

управление модулями курсов и их содержимым

Мы создадим систему, которая позволит управлять модулями и их содержи-

мым. Для этого нам понадобятся формы для нескольких модулей курса и не-

331

скольких объектов содержимого для модуля. Внутри модуля и курса вложенные объекты должны быть расположены в определенном порядке.

**Использование наборов форм для модулей курсов**

В Django реализован механизм работы с несколькими формами на одной странице. Такая **группа, состоящая из нескольких форм, называется набором форм, или формсетом**. Набор форм может обработать несколько объектов типа Form или ModelForm. Все они отправляются на сервер за один раз, а формсет отвечает за то, чтобы определить их количество, понять, сколько форм нужно отображать при редактировании объектов, установить ограничение максимального количества создаваемых объектов и валидацию.

У класса формсета есть метод is\_valid() , позволяющий проверить корректность

всех входящих в него форм за один раз. Кроме того, есть возможность предоставить начальные данные, которыми будут заполнены формы, и указать, сколько пустых форм для каждого типа объектов следует отображать на странице.

Более подробная информация о формсетах приведена в документации на

страницах <https://docs.djangoproject.com/en/2.0/topics/forms/forms>

и https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/#model(-form sets - для модельных форм).

Так как курс может состоять из нескольких модулей, имеет смысл применить набор форм.

**Создайте файл forms.py в папке приложения courses и добавьте в него такой код:**

from django import forms

from django.forms.models import inlineformset\_factory

from .models import Course, Module

ModuleFormSet = inlineformset\_factory(Course,

Module,

fields=['title', 'description'],

extra=2,

can\_delete=True)

Это набор форм ModuleFormSet. Мы формируем его с помощью фабричной

функции Djangion lineformset\_factory(). Получаем набор форм, когда объекты

одного типа, модули, будут связаны с объектом другого типа, курсами.

Мы также передали в фабричную функцию следующие аргументы:

* fields – поля, которые будут добавлены для каждой формы набора;
* extra – количество дополнительных пустых форм модулей;
* can\_delete. Если установить его в True, Django добавит для каждой формы

модулей чекбокс, с помощью которого можно отметить объект к удалению.

**Отредактируйте файл views.py приложения courses** и добавьте в него такой

код:

from django.shortcuts import redirect, get\_object\_or\_404

from django.views.generic.base import TemplateResponseMixin, View

332

from .forms import ModuleFormSet

class CourseModuleUpdateView(TemplateResponseMixin, View): # page 332

template\_name = 'courses/manage/module/formset.html'

course = None

def get\_formset(self, data=None): # page 332

return ModuleFormSet(instance=self.course,

data=data)

def dispatch(self, request, pk): # page 332

self.course = get\_object\_or\_404(Course,

id=pk,

owner=request.user)

return super(CourseModuleUpdateView, self).dispatch(request, pk)

def get(self, request, \*args, \*\*kwargs): # page 332

formset = self.get\_formset()

return self.render\_to\_response({'course': self.course, 'formset': formset})

def post(self, request, \*args, \*\*kwargs): # page 332

formset = self.get\_formset(data=request.POST)

if formset.is\_valid():

formset.save()

return redirect('manage\_course\_list')

return self.render\_to\_response({'course': self.course,

'formset': formset})

Класс CourseModuleUpdateView обрабатывает действия, связанные с набором форм по сохранению, редактированию и удалению модулей для конкретного курса. Этот обработчик наследуется от таких классов:

* TemplateResponseMixin– примесь, которая добавит формирование HTML-

шаблона и вернет его в качестве ответа на запрос. Она использует шаблон, имя которого задано в атрибутe tempеlate\_name, и добавляет в дочерние классы метод render\_to\_response(), который сформирует результирующую страницу;

* View – базовый класс для обработчиков Django.

В этом обработчике мы описали четыре метода:

* get\_formset() – метод, позволяющий избежать дублирования кода, кото-

рый отвечает за формирование набора форм;

* dispatch() – метод, определенный в базовом клаVсiсeеw. Он принимает

объект запроса и его параметры и пытается вызвать метод, который соответствует HTTP-методу запроса. Если запрос отправлен с помощью GET,

его обработка будет делегирована методу get() обработчика; если POST–

то методу post(). Внутри функции мы пытаемся получить объект типа

Course с помощью get\_object\_or\_404(), т. к. курс необходим как для GET -,

так и для POST-запросов, а затем сохраняем его в атрибут класса course;

* get() – метод, обрабатывающий GET-запрос. Мы создаем пустой набор

форм ModuleFormSet и отображаем его в шаблоне с данными курса, используя для этого метод родительского класса render \_to\_response();

333

* post() – метод, обрабатывающий POST-запросы. При этом выполняются

следующие шаги:

- создается набор форм ModuleFormSet по отправленным данным;

- происходит проверка данных с помощью метода is\_valid() набора

форм;

- если все формы заполнены корректно, сохраняем объекты, вызвав

метод save(). На этом этапе в базу данных будут сохранены не только данные курса, но и все изменения модулей. Затем перенаправляем пользователя на страницу по URLʼ у manage\_course\_list. Если хотя бы одна форма набора заполнена некорректно, формируем страницу с отображением ошибок.

**Отредактируйте файл urls .py приложения courses** и добавьте в него такой

шаблон:

path('<pk>/module/', # page 333

views.CourseModuleUpdateView.as\_view(),

name='course\_module\_update'),

**Создайте новую папку в каталоrе couгses/manage/и назовите ее module.**

**Добавьте файл courses/manage/module/formset.html** с таким содержимым:

{% extends "base.html" %}

{% block title %}

Edit "{{ course.title }}"

{% endblock %}

{% block content %}

<h1>Edit "{{ course.title }}"</h1>

<div class="module">

<h2>Course modules</h2>

<form action="" method="post">

{{ formset }}

{{ formset.management\_form }}

{% csrf\_token %}

<input type="submit" class="button" value="Save modules">

</form>

</div>

{% endblock %}

В этот фрагмент мы добавил и < тег form >, в котором будет отображена переменная контекста formset. Мы также добавили форму управления формсетом с помощью запис и { { formset.management\_form } }. Она невидима для пользователя и содержит служебную информацию о количестве форм в наборе. Работа завершена.

Вы можете убедиться, как легко добавить отображение формсета на странице.

**Отредактируйте файл соurses/manage/course/list.html,** добавьте в него ссылку

на страницу редактирования модулей курса:

<a href="{% url "course\_edit" course.id %}">Edit</a>

<a href="{% url "course\_delete" course.id %}">Delete</a>

<a href="{% url "course\_module\_update" course.id %}">Edit modules</a>

334

Теперь откройте в браузере страницу http://127.0.0.1:8000/course/mine/, создайте новый курс и нажмите ссылку Edit modules. Вы увидите такой набор форм:



Рис. 10.9 Форма добавления модуля для курса из формсета

Набор содержит форму для каждого объекта Module, связанного с курсом.

После заполненных форм будут отображены две пустые, т. к. мы указали параметр extra= 2 для ModuleFormSet. Когда вы сохраняете такой формсет, Django записывает в базу данных все модули, которые заполнены в формах.

Добавление содержимого в модуль

Для каждого модуля владельцы курсов могут задавать сколько угодно объектов содержимого: текста, файлов, картинок или видео. Для этого необходимо добавить обработчики, которые позволят создать, изменить или удалить содержимое модуля. На этот раз мы найдем более универсальное решение, т. к. будем работать с объектами четырех моделей.

335

**Вставьте в файл views.py приложения** courses такой код:

from django.forms.models import modelform\_factory

from django.apps import apps

from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View): # 335

module = None

model = None

obj = None

template\_name = 'courses/manage/content/form.html'

def get\_model(self, model\_name):

if model\_name in ['text', 'video', 'image', 'file']:

return apps.get\_model(app\_label='courses',

model\_name=model\_name)

return None

def get\_form(self, model, \*args, \*\*kwargs): # 335

Form = modelform\_factory(model, exclude=['owner',

'order',

'created',

'updated'])

return Form(\*args, \*\*kwargs)

def dispatch(self, request, module\_id, model\_name, id=None): # 335

self.module = get\_object\_or\_404(Module,

id=module\_id,

course\_\_owner=request.user)

self.model = self.get\_model(model\_name)

if id:

self.obj = get\_object\_or\_404(self.model,

id=id,

owner=request.user)

return super(ContentCreateUpdateView,

self).dispatch(request, module\_id, model\_name, id)

Это первая часть обработчика ContentCreateUpdateView. Код, который мы опи-

сали выше, позволит создавать и редактировать содержимое различных типов.

В обработчик добавлены такие методы:

* get\_model() – возвращает класс модели по переданному имени. Допусти-

мые значения –Text, Video, Image и File . Мы обращаемся к модулю apps

Django, чтобы получить класс модели. Если его не удалось найти по переданному имени, возвращаем None;

* get\_form() – создает форму в зависимости от типа содержимого с помощью функции modelform\_factory(). Так как модели Text, Video, Imageи File содержат общие поля, исключим их из формы, чтобы пользователь заполнял только поле непосредственного содержимого (файл, текст, картинку или видео);
* dispatch() – получает приведенные ниже данные из запроса и создает

соответствующие объекты модуля, модели содержимого:

336

- module\_id – идентификатор модуля, к которому привязано содержимое;

- model\_name – имя модели содержимого;

- id – идентификатор изменяемого объекта.

**Добавьте методы get() и post() в класс ContentCreateUpdateView:**

def get(self, request, module\_id, model\_name, id=None): # page 336

form = self.get\_form(self.model, instance=self.obj)

return self.render\_to\_response({'form': form, 'object': self.obj})

def post(self, request, module\_id, model\_name, id=None): # page 336

form = self.get\_form(self.model,

instance=self.obj,

data=request.POST,

files=request.FILES)

if form.is\_valid():

obj = form.save(commit=False)

obj.owner = request.user

obj.save()

if not id:

# Создаем новый объект.

Content.objects.create(module=self.module, item=obj)

return redirect('module\_content\_list', self.module.id)

return self.render\_to\_response({'form': form, 'object': self.obj})

Они выполняют следующие действия:

* get() – извлекает из GET-параметров запроса данные. Формирует модельные формы для объектов Text, Video, Image или File , если объект редактируется, т. е. указан self.obj . В противном случае мы отображаем пустую форму для создания объекта;
* post() – обрабатывает данные POST-запроса, для чего создает модельную форму и валидирует ее. Если форма заполнена корректно, создает новый объект, указав текущего пользователя request.user, владельцем.

Если в запросе был передан ID, значит, объект изменяют, а не создают.

**Отредактируйте файл urls.py приложенияo curses и добавьте такие шаблоны**

**URL’ов:**

path('module/<int:module\_id>/content/<model\_name>/create/', # page 336

views.ContentCreateUpdateView.as\_view(),

name='module\_content\_create'),

path('module/<int:module\_id>/content/<model\_name>/<id>/', # page 336

views.ContentCreateUpdateView.as\_view(),

name='module\_content\_update'),

Мы добавили два шаблона, которые будут использоваться для:

* module\_content\_create – дает доступ к обработчику создания нового со-

держимого и привязки его к модулю. Он содержит параметры module\_ id

и model\_name ID. используется для получения модуля, которому добавляют содержимое, а имя модели нужно, чтобы определить тип содержимого;

337

* module\_content\_update – дает доступ к обработчику редактирования содержимого. По аналогии с первым шаблоном принимает два параметра(module\_id и module\_id,) которые использует таким же образом.

**Создайте новую папку в каталоге courses/manage/ и назовите ее content.**

**Добавьте файл courses/manage/content/form.html с таким содержимым:**

{% extends "base.html" %}

{% block title %}

{% if object %}

Edit content "{{ object.title }}"

{% else %}

Add a new content

{% endif %}

{% endblock %}

{% block content %}

<h1>

{% if object %}

Edit content "{{ object.title }}"

{% else %}

Add a new content

{% endif %}

</h1>

<div class="module">

<h2>Course info</h2>

<form action="" method="post" enctype="multipart/form-data">

{{ form.as\_p }}

{% csrf\_token %}

<p><input type="submit" value="Save content"></p>

</form>

</div>

{% endblock %}

Это шаблон для обработчика ContentCreateUpdateView. При формировании

HTML-страницы мы проверяем, передан ли объект содержимого в переменной object. Если передан, отображаем форму редактирования, если нет – форму создания.

Вы могли заметить, что мы указали атрибут enctype="multipart/form-data" для

тега <form>. Это необходимо, чтобы при загрузке картинок и файлов их содержимое было передано на сервер.

Запустите сервер для разработки и перейдите на страницу <http://127.0.0.1:8000/course/mine/>

Нажмите на ссылку Edit modules на странице курса и создайте модуль.

Откройте консоль Python с помощью команды

python manage.py shell

и запросите ID самого последнего модуля:

>>> from courses.models import Module

>>> Module.objects.latest('id').id

6

338

Запустите сервер для разработки и перейдите по адресу вида <http://127.0.0.1:8000/course/module/6/content/image/create/>, заменив ID модуля на тот, который получили благодаря выполнению предыдущего шага. Вы увидите форму добавления картинки:

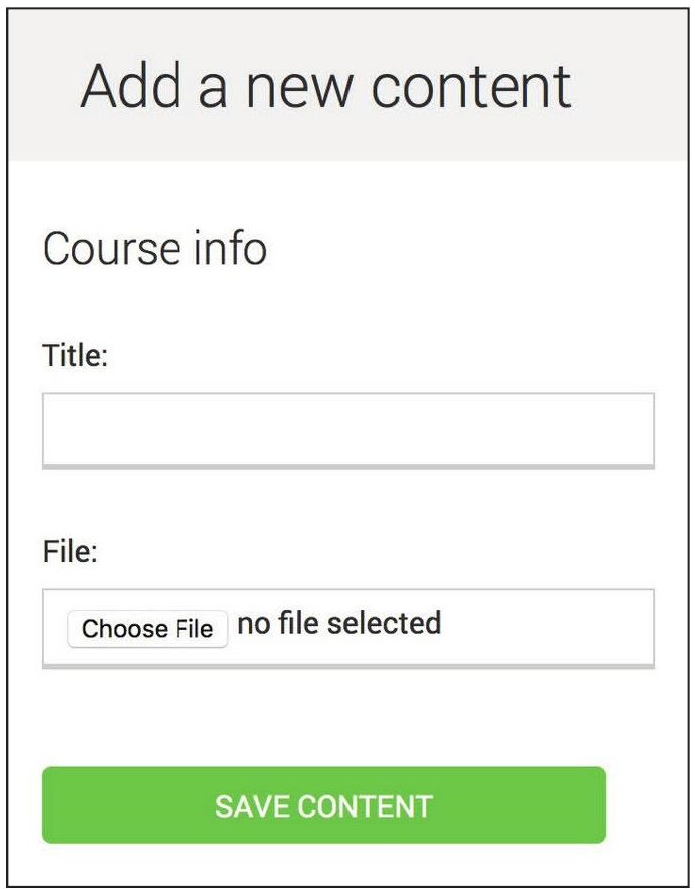


Рис. 10.10 Форма создания содержимого модуля

Пока не отправляйте ее. Если вы попытаетесь это сделать, получите ошибку,

потому что мы пока не определили шаблон URLʼа module\_content\_list. Сделаем

это чуть-чуть позже.

На текущий момент нам осталось добавить обработчик для удаления содержимого модулей.

**Откройте файл views.py приложения courses и добавьте** такой класс:

class ContentDeleteView(View): # 338

def post(self, request, id):

content = get\_object\_or\_404(Content,

id=id,

module\_\_course\_\_owner=request.user)

module = content.module

content.item.delete()

content.delete()

return redirect('module\_content\_list', module.id)

Обработчик ContentDeleteView получает объект типа Content по переданному

ID и удаляет соответствующий объект модeлиTеxt, Video, Image или File , после

339

чего ликвидирует объект Content. При успешном завершении действия перенаправляет пользователя на страницу по URLʼу с именем module\_content\_list.

Добавьте шаблон URLʼа для этого обработчика.

**В файл urls.py приложения courses** вставьте такие строки:

path('content/<int:id>/delete/', # page 339

views.ContentDeleteView.as\_view(),

name='module\_content\_delete'),

Теперь владельцы курсов смогут добавлять, изменять и удалять содержимое

модулей.

**Управление модулями и их содержимым**

Мы создали обработчики для модулей и содержимого. Давайте добавим возможность отображать список всех модулей курса и всего содержимого модуля.

**Отредактируйте файл views.py приложения courses** и добавьте в него такой

код:

class ModuleContentListView(TemplateResponseMixin, View): # page 339

template\_name = 'courses/manage/module/content\_list.html'

def get(self, request, module\_id): # page 339

module = get\_object\_or\_404(Module,

id=module\_id,

course\_\_owner=request.user)

return self.render\_to\_response({'module': module})

Это обработчик ModuleContentListView. Он получает из базы данных модуль по

переданному ID и генерирует для него страницу подробностей.

**Добавьте в файл urls.py приложения courses** такие строки:

path('module/<int:module\_id>/', # page 339

views.ModuleContentListView.as\_view(),

name='module\_content\_list'),

**Music\_training/courses/templates/courses/manage/module/content\_list.html**

**Создайте новый файл, content\_list.html, в папке templates/courses/manage/module/ и вставьте в него следующий код:**

{% extends "base.html" %}

{% block title %}

Module {{ module.order|add:1 }}: {{ module.title }}

{% endblock %}

{% block content %}

{% with course=module.course %}

<h1>Course "{{ course.title }}"</h1>

<div class="contents">

<h3>Modules</h3>

<ul id="modules">

{% for m in course.modules.all %}

<li data-id="{{ m.id }}" {% if m == module %}

class="selected"{% endif %}>

340

<a href="{% url "module\_content\_list" m.id %}">

<span>

Module <span class="order">{{ m.order|add:1 }}</span>

</span>

<br>

{{ m.title }}

</a>

</li>

{% empty %}

<li>No modules yet.</li>

{% endfor %}

</ul>

<p><a href="{% url " courses:course\_module\_update" course.id %}">

Edit modules</a></p>

</div>

<div class="module">

<h2>Module {{ module.order|add:1 }}: {{ module.title }}</h2>

<h3>Module contents:</h3>

<div id="module-contents">

{% for content in module.contents.all %}

<div data-id="{{ content.id }}">

{% with item=content.item %}

<p>{{ item }}</p>

<a href="#">Edit</a>

<form action="{% url ‘courses:module\_content\_delete’ content.id %}"

method="post">

<input type="submit" value="Delete">

{% csrf\_token %}

</form>

{% endwith %}

</div>

{% empty %}

<p>This module has no contents yet.</p>

{% endfor %}

</div>

<h3>Add new content:</h3>

<ul class="content-types">

<li><a href="{% url "module\_content\_create" module.id "text" %}">

Text</a></li>

<li><a href="{% url "module\_content\_create" module.id "image" %}">

Image</a></li>

<li><a href="{% url "module\_content\_create" module.id "video" %}">

Video</a></li>

<li><a href="{% url "module\_content\_create" module.id "file" %}">

File</a></li>

</ul>

</div>

{% endwith %}

{% endblock %}

Этот шаблон будет формировать все модули курса и их содержимое. Мы обращаемся к списку модулей, чтобы отобразить их в боковой панели. Затем проходим

341

по списку содержимого каждого модуля, обращаяcь к сontent.item, чтобы

получить соответствующий объект одного из типов Text, Video, Image или File .

Также отображаем ссылку для добавления содержимого к модулю.

Мы хотим знать, какому типу принадлежит каждый объект содержимого

item, т. к. нам необходимо формировать ссылку, основываясь на имени модели.

Кроме этого, содержимое разных типов по-разному формируется в HTML, поэ-

тому мы могли бы обратиться к опциям класса Meta модели через атрибут \_meta.

Но шаблонизатор Django не допускает такую запись, и вы получите ошибку,

если используете этот атрибут в шаблоне. Для решения данной проблемы нам

понадобится собственный шаблонный фильтр.

**Создайте такую структуру папок и файлов в каталоге прилoжения cоurses:**

templatetags/

\_\_init\_\_.py

course.py

**Добавьте в файл course.py** фрагмент кода:

from django import template

register = template.Library() # 341

@register.filter

def model\_name(obj): # 341

try:

return obj.\_meta.model\_name

except AttributeError:

return None

Только что мы определили шаблонный фильтр model\_name, который можно применить в шаблонах, чтобы получить имя модели объекта. Для этого используйте запись вида object|model\_name.

**Отредактируйте файл templates/courses/manage/module/content\_list.html** и добавьте такую строку после тега { % extends %:}

{% load course %}

Мы подключили модуль course, и только что созданный нами шаблонный

фильтр теперь доступен.

Найдите следующие строки в файле:

<p>{{ item }}</p>

<a href="#">Edit</a>

Замените их таким образом:

<p>{{ item }} ({{ item|model\_name }})</p>

<a href="{% url "module\_content\_update" module.id item|model\_name item.id

%}">Edit</a>

Теперь мы показываем ссылку на редактирование объекта нужного типа.

**Измените файл courses/manage/course/list.html,** чтобы ссылка на URL module\_content\_list приобрела указанный ниже вид:

342

<a href="{% url ‘courses:course\_module\_update’ course.id %}">Edit modules</a>

{% if course.modules.count > 0 %}

<a href="{% url ‘courses:module\_content\_list’ course.modules.first.id %}">

Manage contents</a>

{% endif %}

Эта ссылка позволит владельцу курса настроить содержимое первого модуля.

Откройте в браузере страницу <http://127.0.0.1:8000/course/mine/> . Кликните

на ссылку Manage contents курса, в котором есть хотя бы один модуль. Вы увидите похожую страницу:

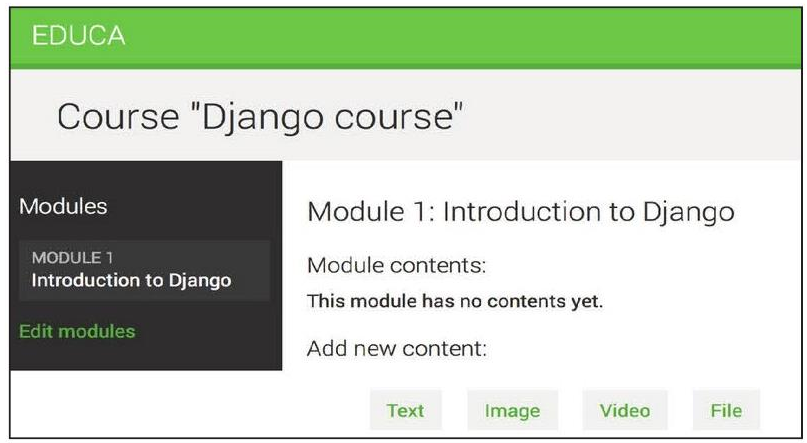


Рис. 10.11 Страница модуля со ссылками на добавление нового содержимого

Когда вы выбираете модуль в левой боковой панели, его содержимое отображается в основном блоке страницы. При этом пользователю доступны ссылки на добавление в модуль нового содержимого. Добавьте несколько объектов и посмотрите на страницу модуля. Содержимое должно будет появиться в основном блоке страницы:

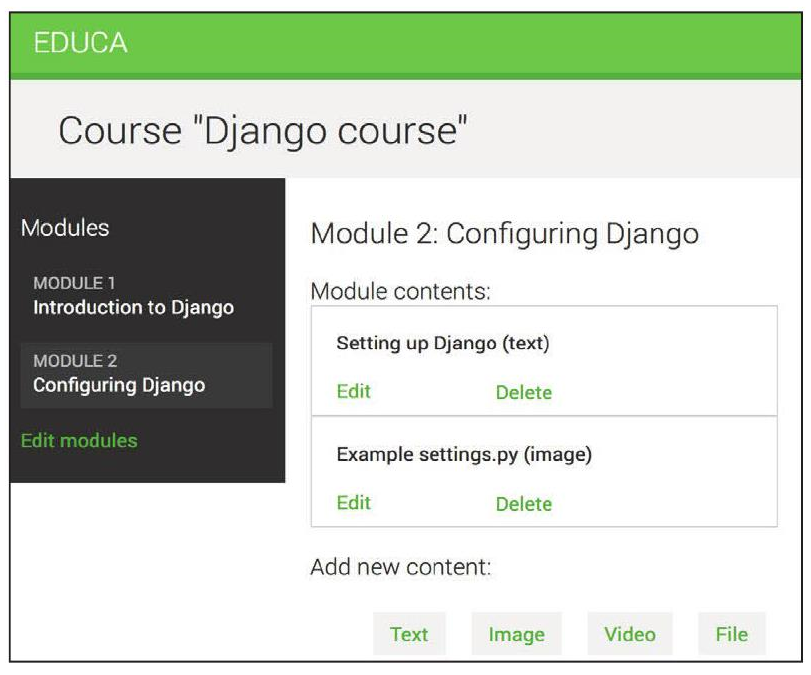


Рис. 10.12 Страница модуля с содержимым

343

**Изменения порядка модулей и их содержимого**

Мы предусмотрели возможность задавать порядок модулей и содержимого. Теперь нам необходимо добавить возможность сделать это через интерфейс. Для этого мы применим drag-n-drop виджет JavaScript, с помощью которого пользователь сможет перетаскивать объекты списка и благодаря этому задавать их порядок. Для сохранения действий в базе данных реализуем AJAX-запрос.

Использование примесей из django-braces

Пакет django-braces – это сторонняя библиотека, которая содержит примеси для Django. Полный список всех примесей можно найти на странице <https://django-braces.readthedocs.io/> .

Мы добавим в проект следующие классы из пакета django-braces:

* CsrfExemptMixin, чтобы отключить проверку CSRF-токена для некоторых

POS-Tзапросов. Нам нужно иметь возможность посылать AJAX-запросы

без генерации токена;

* JsonRequestResponseMixin – преобразует данные запроса в JSON и сериализует ответ в JSON, возвращая его с типом application/json.

Установите пакет django-braces с помощью pip:

**pip install django-braces==1.13.0**

Нам нужен обработчик, который будет получать новый порядок модулей

курса в формате JSON.

**Отредактируйте файл view.py приложения courses и добавьте в него такой код:**

from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class ModuleOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View): # 343

def post(self, request): # page 343

for id, order in self.request\_json.items():

Module.objects.filter(id=id,

course\_\_owner=request.user).update(order=order)

return self.render\_json\_response({'saved': 'OK'})

Это обработчик ModuleOrderView для модулей.

Теперь добавим аналогичный обработчик для содержимого модулей.

**Вставьте в файл views.py фрагмент:**

class ContentOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View): # 343

def post(self, request): # page 343

for id, order in self.request\_json.items():

Content.objects.filter(id=id,

module\_\_course\_\_owner=request.user).update(order=order)

return self.render\_json\_response({'saved': 'OK'})

**Добавьте в файл urls.py приложения courses шаблоны URLʼов для этих обработчиков:**

path('module/order/', # page 343

views.ModuleOrderView.as\_view(),

344

name='module\_order'),

path('content/order/', # page 344

views.ContentOrderView.as\_view(),

name='content\_order'),

Наконец, давайте подключим виджет в шаблоне.

Мы будем использовать библиотеку jQuery UI. Она основывается на jQuery и предоставляет плагины для интерфейса (различных виджетов и визуальных эффектов). Для начала необходимо загрузить эту библиотеку.

**Откройте файл templates/base.html приложения courses** и добавьте подключение скрипта после jQuery таким образом:

<script

src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<script

src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js"></script>

**Теперь отредактируйте файл courses/manage/module/content\_list.html** и до-

бавьте следующий фрагмент:

{% block domready %}

$('#modules').sortable({

stop: function(event, ui) {

modules\_order = {};

$('#modules').children().each(function(){

// Обновляем поле порядкового номера.

$(this).find('.order').text($(this).index() + 1);

// Связываем порядковый номер с идентификатором объекта.

modules\_order[$(this).data('id')] = $(this).index();

});

$.ajax({

type: 'POST',

url: '{% url "courses:module\_order" %}',

contentType: 'application/json; charset=utf-8',

dataType: 'json',

data: JSON.stringify(modules\_order)

});

}

});

$('#module-contents').sortable({

stop: function(event, ui) {

contents\_order = {};

$('#module-contents').children().each(function(){

// Связываем порядковый номер с идентификатором объекта.

contents\_order[$(this).data('id')] = $(this).index();

});

$.ajax({

type: 'POST',

url: '{% url "courses:content\_order" %}',

contentType: 'application/json; charset=utf-8',

dataType: 'json',

345

data: JSON.stringify(contents\_order),

});

}

});

{% endblock %}

Этот код размещен в блоке { % block domready%,} поэтому будет выполняться после построения DOM-дерева страницы. Таким образом, мы гарантируем, что все HTML-элементы, необходимые для работы виджета, уже будут созданы на странице. Мы определили объект списка модулей из боковой панели как sortable. То же самое сделали для списка содержимого каждого модуля. В этом фрагменте кода мы выполняем следующие действия:

1) определяем списки как sortable-объекты, используя селекторы jQuery

вида #modules;

2) определяем функцию для события stop, срабатывающего каждый раз,

когда пользователь заканчивает сортировку объектов;

3) создаем пустой словарь modules\_order. Ключами являются идентификаторы модулей, а значениями – их порядковые номера в списке;

4) проходим по списку модулей, пересчитываем порядок для каждого из

них и получаем ID, обратившись к атрибуту data-id . Добавляем пару

«идентификатор – номер» в словарь modules\_order;

5) выполняем AJAX-запрос методом POST на обработчик по URLʼу с именем

content\_order. В качестве параметра передаем переменную nmodules \_order

в виде JSON. Обработчик ModuleOrderView сохраняет изменения в базу данных.

Элемент sortable для содержимого модуля работает аналогично.

**Перейдите в браузер и перезагрузите страницу**. Теперь вы можете перетаскивать модули в боковой панели, меняя их порядок в рамках курса:

<http://127.0.0.1:8000/course/mine/>

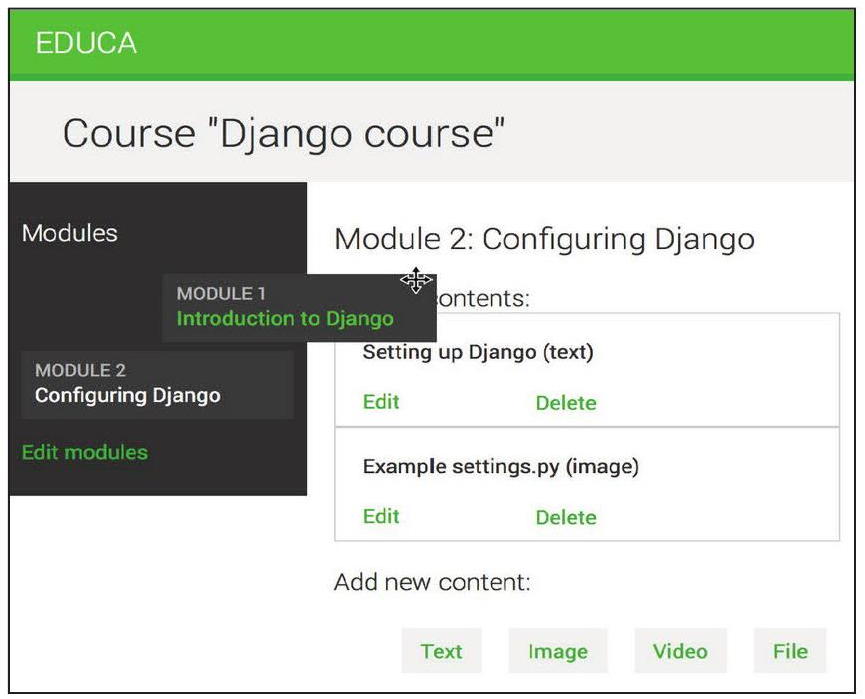


Рис. 10.13 Изменение порядка модулей курса

346

Поздравляем! Теперь владельцы курсов могут менять порядок модулей и их

содержимого

Pезюме.

Благодаря этой главе вы узнали, как создать простую, но в то же время достаточно мощную CMS. Вы применили наследование моделей и реализовали собственный класс для поля модели, а также научились работать с наборами форм для различных типов объектов.

В следующей главе вы узнаете, как создавать системы для регистрации студентов курсов, будете формировать различный вид страниц для разных типов объектов и узнаете, как работать с подсистемой кеширования Django.

**Глава 11**

**Отображение**

**и кеширование**

**содержимого курсов**

В предыдущей главе вы применили такой прием, как наследование моделей,

и реализовали с помощью обобщенных отношений платформу для онлайн-

обучения с настраиваемыми модулями и содержимым. Для этого вы использовали обработчики в виде классов, наборы форм и AJAX-запросы. В этой главе мы рассмотрим следующие темы:

* создание обработчиков для показа содержимого курсов студентам;
* добавление системы регистрации пользователей;
* управление участием студентов в курсах;
* формирование вида содержимого курсов в зависимости от типа;
* кеширование с помощью стороннего приложения.

Начнем знакомство с создания каталога курсов, чтобы студенты могли искать интересные им и записываться.

**Отображение курсов**

Для каталога нам понадобятся два обработчика:

* для списка курсов с возможностью фильтрации;
* для страницы подробностей курса.

**Отредактируйте файл views.py приложения courses** и добавьте в него такой

код:

from django.db.models import Count

from .models import Subject

class CourseListView(TemplateResponseMixin, View): # page 347

model = Course

template\_name = 'courses/course/list.html'

def get(self, request, subject=None): # page 347

348

subjects = Subject.objects.annotate(

total\_courses=Count('courses'))

courses = Course.objects.annotate(total\_modules=Count('modules'))

if subject:

subject = get\_object\_or\_404(Subject, slug=subject)

courses = courses.filter(subject=subject)

return self.render\_to\_response({'subjects': subjects,

'subject': subject,

'courses': courses})

Это обработчик CourseListView. Он является наследником примеси TemplateResponseMixin и класса View. При обработке запроса на получение курсов мы выполняем следующие действия:

1) получаем список всех предметов, добавляя количество курсов по каждому из них. Для этого применяем метод annotate() QuerySetʼа и функцию агрегации Count();

2) получаем все доступные курсы, включая количество модулей для каждого из них;

3) если в URLʼе задан слаг предмета, получаем объект предмета и фильтруем список курсов по нему;

4) для формирования результата используем метод render\_ to\_response() из

примеси TemplateResponseMixin.

Давайте создадим обработчик, который будет показывать страницу курса.

**Добавьте такой фрагмент кода в файл views. py:**

from django.views.generic.detail import DetailView

class CourseDetailView(DetailView): # page 348 обработчик, который будет

# показывать страницу курса

model = Course

template\_name = 'courses/course/detail.html'

Этот обработчик наследуется от стандартного класса DetailDjangoView, для

отображения объекта конкретной модели. Мы указали два атрmибута: lode

и template\_name. При обработке запроса Django ожидает, что в URL будет передан идентификатоp (рk) объекта, по которому можно его получить. Затем формирует результат в виде HTML-страницы, сгенерированной из шаблона с именем template\_name. В контекст шаблона добавляется переменная – объект модели.

**Отредактируйте файл urls.py проекта educa** и добавьте в него подключение

шаблонов URL’а, как показано ниже:

from courses.views import CourseListView

urlpatterns = [

# ...

path('', CourseListView.as\_view(), name='course\_list'), # page 348

]

Мы добавили шаблон с именем course\_list в файл urls.py проекта, т. к. хотим

обращаться к нему без префикса приложения, по адресу: http://127.0.0.1:8000/.

Для всех остальных шаблонов, которые будем размещать в соответствующем

файле приложения courses, в URL будет добавляться префикс/course/.

349

**Откройте файл urls.py приложения courses и добавьте в него следующие**

**строки:**

path('subject/<slug:subject>)/', # page 349

views.CourseListView.as\_view(),

name='course\_list\_subject'),

path('<slug:slug>/', # page 349

views.CourseDetailView.as\_view(),

name='course\_detail'),

Мы определили два шаблона URLʼов:

* course\_list\_subject – для отображения всех курсов по выбранному пред-

мету;

* course\_detail – для страницы подробного описания курса.

Теперь давайте создадим HTML-шаблоны для обработчик в CourseоListView

и CourseDetailView.

**Создайте в каталоге templates/courses / приложения courses** такую структуру папок и файлов:

course/

list.html

detail.html

**Вставьте в файл courses/course/list.html** следующий фрагмент:

{% extends "base.html" %}

{% block title %}

{% if subject %}

{{ subject.title }} courses

{% else %}

All courses

{% endif %}

{% endblock %}

{% block content %}

<h1>

{% if subject %}

{{ subject.title }} courses

{% else %}

All courses

{% endif %}

</h1>

<div class="contents">

<h3>Subjects</h3>

<ul id="modules">

<li {% if not subject %}class="selected"{% endif %}>

<a href="{% url "course\_list" %}">All</a>

</li>

{% for s in subjects %}

<li {% if subject == s %}class="selected"{% endif %}>

<a href="{% url "courses:course\_list\_subject" s.slug %}">

{{ s.title }}

350

<br><span>{{ s.total\_courses }} courses</span>

</a>

</li>

{% endfor %}

</ul>

</div>

<div class="module">

{% for course in courses %}

{% with subject=course.subject %}

<h3><a href="{% url "courses:course\_detail" course.slug %}">

{{ course.title }}</a></h3>

<p>

<a href="{% url "courses:course\_list\_subject" subject.slug %}">

{{ subject }}</a>.

{{ course.total\_modules }} modules.

Instructor: {{ course.owner.get\_full\_name }}

</p>

{% endwith %}

{% endfor %}

</div>

{% endblock %}

Этот шаблон будет формировать список всех курсов. Мы создаем список

предметов Subject. Нажатие на любой его элемент будет перенаправлять пользователя на страницу отфильтрованных курсов, course \_list\_subject. Для выделения предмета, если он выбран, используем CSS-класс selected .

В заключение проходим по каждому объекту Course и показываем количество модулей и имя преподавателя.

Запустите сервер для разработки и перейдите на страницу [**http://127.0.0.1:8000/**](http://127.0.0.1:8000/) .

Вы увидите похожую страницу:

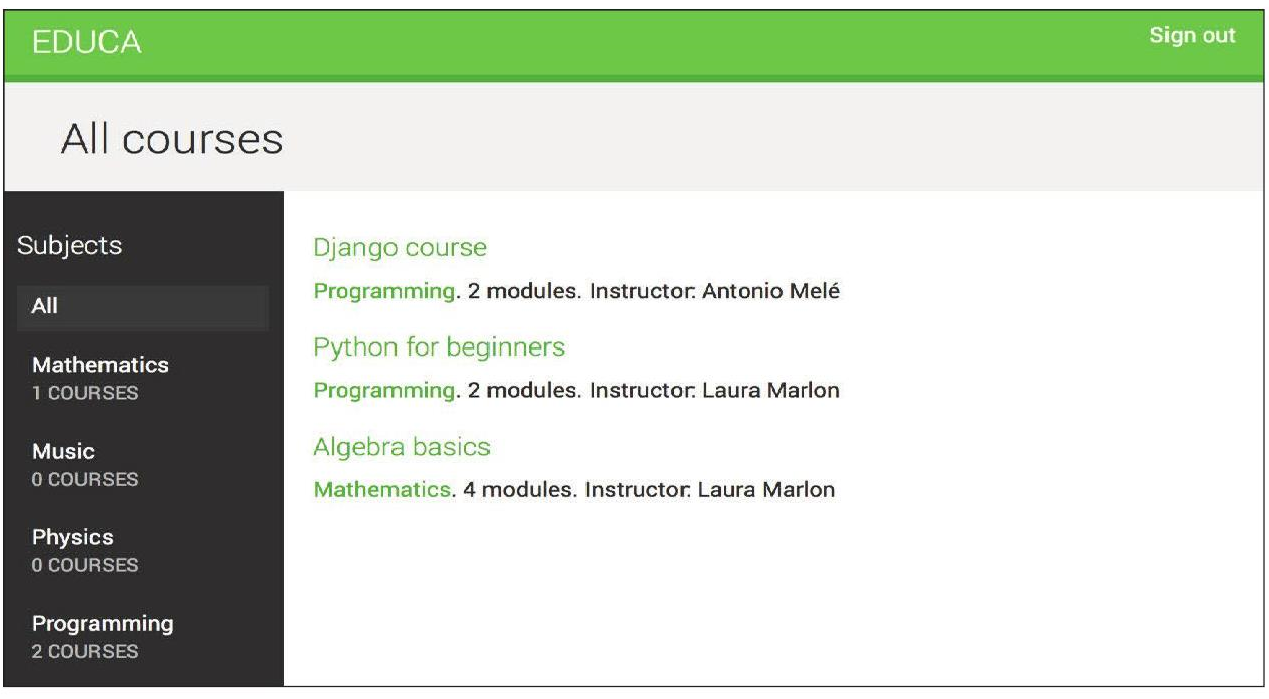


Рис. 11.1 Список курсов

351

В левой боковой панели перечислены все предметы, включая количество

курсов. Вы можете кликнуть на любой из них и попадете на страницу с отфильтрованным списком курсов.

**Отредактируйте файл courses/course/detail.html и добавьте такой код:**

{% extends "base.html" %}

{% block title %}

{{ object.title }}

{% endblock %}

{% block content %}

{% with subject=course.subject %}

<h1>

{{ object.title }}

</h1>

<div class="module">

<h2>Overview</h2>

<p>

<a href="{% url "courses:course\_list\_subject" subject.slug %}">

{{ subject.title }}</a>.

{{ course.modules.count }} modules.

Instructor: {{ course.owner.get\_full\_name }}

</p>

{{ object.overview|linebreaks }}

</div>

{% endwith %}

{% endblock %}

В этом шаблоне мы показываем подробную информацию о курсе.

**Откройте в браузере http://127.0.0.1:8000/** и кликните на какой-нибудь курс. Вы должны будете увидеть страницу с такой структурой:

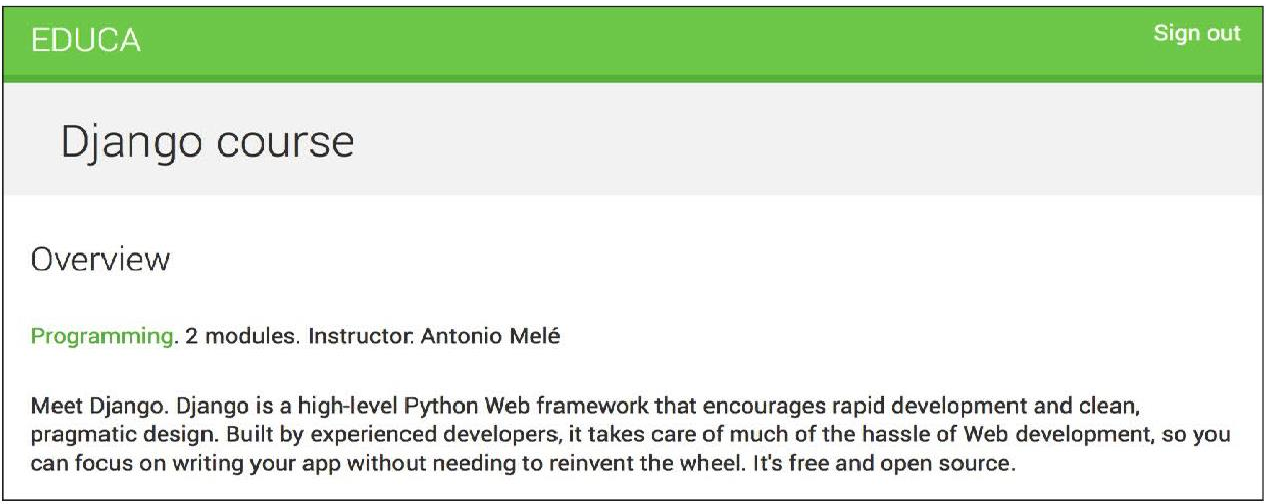


Рис. 11.2 Страница описания курса

Мы создали публичные страницы для доступа всех пользователей к курсам.

Далее необходимо добавить возможность регистрации на сайте и записи на курс.

352

**Добавление регистрации обучающихся**

Создайте новое приложение, выполнив в консоли следующую команду:

python manage.py startapp students

у нас:

**(mus\_venv) PS E:\Django\_Projects \Music\_training \Music\_training> django-admin manage.py startapp students**

**Отредактируйте файл settings.py проекта educa** и добавьте созданное при-

ложение в список INSTALLED\_APPS:

INSTALLED\_APPS = [

# ...

'students.apps.StudentsConfig', # page 352 регистрация обучающихся

]

**Обработка регистрации обучающихся в системе**

**Откройте файл views.py приложения students и** вставьте в него следующий

фрагмент:

from django.urls import reverse\_lazy

from django.views.generic.edit import CreateView

from django.contrib.auth.forms import UserCreationForm

from django.contrib.auth import authenticate, login

class StudentRegistrationView(CreateView): # page 352 Обработка регистрации обучающихся в системе

template\_name = 'students/student/registration.html'

form\_class = UserCreationForm

success\_url = reverse\_lazy('student\_course\_list')

def form\_valid(self, form): # page 352

result = super(StudentRegistrationView, self).form\_valid(form)

cd = form.cleaned\_data

user = authenticate(username=cd['username'],

password=cd['password1'])

login(self.request, user)

return result

Это обработчик регистрации студентов на сайте. Мы используем специальный класс CreateView, который предоставляет методы создания объектов заданной модели. Также мы определяем несколько атрибутов модели:

* template\_name – имя HTML-шаблона, который будет использоваться;
* form\_class – класс формы для создания объекта. Указанный класс должен

быть наследником ModelForm. Мы указали форму UserCreationForm для создания объектов модели User;

* success\_url – адрес, на который пользователь будет перенаправлен после

успешной обработки формы регистрации. Мы получаем URL по имени

student\_course\_list, но создадим его чуть позже.

Метод form\_valid() обработчика будет выполняться при успешной валидации формы. Он должен возвращать объект HTTP-ответа. Мы переопределили его в нашем обработчике, чтобы после регистрации автоматически авторизовать пользователя на сайте.

353

**Добавьте новый файл urls .py, в папку приложения students** и вставьте в него

такой код:

from django.urls import path

from . import views

urlpatterns = [ # page 353

path('register/',

views.StudentRegistrationView.as\_view(),

name='student\_registration'),

]

Затем подключите созданный файл в основной конфигурации URLʼов проекта educa.

**Для этого добавьте в файл urls. py проекта** **Music\_training** строку, как показано ниже:

urlpatterns = [

# ...

path('students/', include('students.urls')), # page 353

]

Создайте в каталоге приложения students следующую структуру папок и файлов:

templates/

students/

student/

registration.html

Теперь вставьте в шаблон students/student/registration.html такой фрагмент:

{% extends "base.html" %}

{% block title %}

Sign up

{% endblock %}

{% block content %}

<h1>

Sign up

</h1>

<div class="module">

<p>Enter your details to create an account:</p>

<form action="" method="post">

{{ form.as\_p }}

{% csrf\_token %}

<p><input type="submit" value="Create my account"></p>

</form>

</div>

{% endblock %}

Запустите сервер разработки и откройте в браузере **http://127.0.0.1:8000/students/register/** . Вы увидите такую страницу:

354

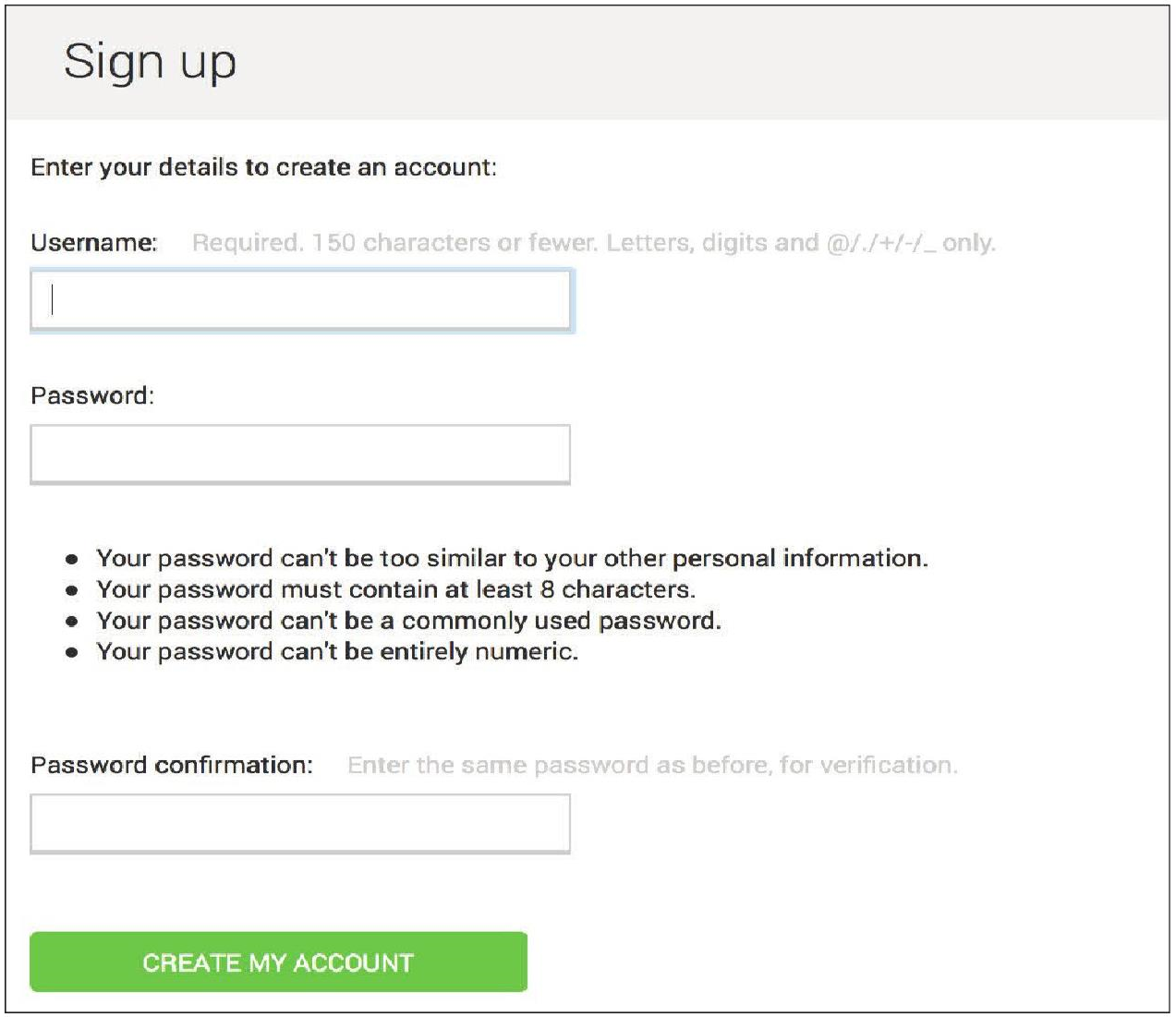


Рис. 11.3 Форма регистрации студентов

Не забудьте, что мы пока не определили URL с именем student\_course\_list,

который задали как атрибуcт sucess\_url обработчика StudentRegistrationView.

Если вы попытаетесь зарегистрироваться, Django не сможет найти этот адрес,

вследствие чего будет сгенерировано исключение. Мы создадим шаблоны

URLʼа и соответствующий обработчик чуть позже.

**Реализация записи на курсы**

После регистрации пользователи должны иметь возможность записываться на

интересные им курсы. Чтобы хранить сведения о том, в каких курсах участвуют студенты, нам понадобится создать связь вида «многие ко многим» между

моделями Course и User.

**Отредактируйте файл models.py приложения courses** и добавьте такую строку

в модель Course:

students = models.ManyToManyField(User, # page 354

related\_name='courses\_joined',

blank=True)

В консоли выполните команду создания миграций:

python manage.py makemigrations

355

Вы увидите такой вывод:

Migrations for 'courses':

courses/migrations/0004\_course\_students.py

- Add field students to course

Затем выполните команду синхронизации с базой данных:

**python manage.py migrate**

Вы увидите вывод, заканчивающийся такой строкой:

Applying courses.0004\_course\_students... OK

Миграции успешно применены, теперь вы можете связывать студентов

и курсы.

Самое время реализовать возможность записываться на курсы.

**Создайте новый файл, forms.py, в папке приложения students** и добавьте

в него следующий фрагмент:

from django import forms

from courses.models import Course

class CourseEnrollForm(forms.Form): # page 355 возможность записываться на курсы

course = forms.ModelChoiceField(queryset=Course.objects.all(),

widget=forms.HiddenInput)

Мы будем использовать эту форму при записи студентов на курсы. В поле

course будет содержаться идентификатор курса, на который происходит запись.

Мы определили его тип как ModelChoiceField и указали виджет HiddenInput, т. к. не хотим, чтобы пользователь видел поле. Эта форма будет использоваться в обработчике CourseDetailView.

**Отредактируйте файл veiws.py приложения students** и вставьте такой фрагмент:

from django.views.generic.edit import FormView

from django.contrib.auth.mixins import LoginRequiredMixin

from .forms import CourseEnrollForm

class StudentEnrollCourseView(LoginRequiredMixin, FormView): # page 355

course = None

form\_class = CourseEnrollForm

def form\_valid(self, form): # page 355

self.course = form.cleaned\_data['course']

self.course.students.add(self.request.user)

return super(StudentEnrollCourseView, self).form\_valid(form)

def get\_success\_url(self): # page 355

return reverse\_lazy('student\_course\_detail', args=[self.course.id])

Это обработчик StudentEnrollCourseView. Он занимается зачислением студентов на курсы. Мы указали родительский класс LoginRequiredMixin, поэтому только авторизованные пользователи смогут записываться. Также родительским

классом является базовый обработчик DjangoFormView, который реализует ра-

356

боту с формой. В атрибуте form\_class мы указали форму CourseEnrollForm, которая при успешной валидации будет создавать связь между студентом и курсом.

Метод get\_success\_url() возвращает адрес, на который пользователь будет

перенаправлен после успешной обработки формы. Этот метод аналогичен

атрибуту success\_url. Мы формируем адрес по шаблону URLʼа с именем student\_course\_detail, который добавим в следующей части этой главы.

**Отредактируйте файл urls .py приложения students** и добавьте такой шаблон:

path('enroll-course/', # page 356

views.StudentEnrollCourseView.as\_view(),

name='student\_enroll\_course'),

Давайте добавим на страницу курса кнопку, нажав на которую, можно будет

записаться на него.

**Откройте файл views.py приложения courses** и отредактируйте класс CourseDetailView, как показано ниже:

from students.forms import CourseEnrollForm

class CourseDetailView(DetailView): # page 356

model = Course

template\_name = 'courses/course/detail.html'

def get\_context\_data(self, \*\*kwargs): # page 356

context = super(CourseDetailView, self).get\_context\_data(\*\*kwargs)

context['enroll\_form'] = CourseEnrollForm(

initial={'course':self.object})

return context

Мы переопределяем метод базового класса get\_context\_data(), чтобы добавить форму в контекст шаблона. Объект формы при этом содержит скрытое

поле с ID курса, поэтому при нажатии кнопки на сервер будут отправлены данные курса и пользователя.

**Music\_training/students/templates/ courses/course/detail.html**

**Откройте файл courses/course/detail.html** и найдите следующую строку:

{{ object.overview|linebreaks }}

Допишите следом такой фрагмент:

{{ object.overview|linebreaks }}

{% if request.user.is\_authenticated %}

<form action="{% url "student\_enroll\_course" %}" method="post">

{{ enroll\_form }}

{% csrf\_token %}

<input type="submit" class="button" value="Enroll now">

</form>

{% else %}

<a href="{% url "student\_registration" %}" class="button">

Register to enroll

</a>

{% endif %}

Это кнопка, по которой студенты смогут записаться на курс, и форма для

отправки данных на URL с именем student\_enroll\_course. Они видны на страни-

357

це только в том случае, если пользователь авторизован на сайте. В противном

случае мы отображаем ссылку на страницу регистрации.

Убедитесь, что сервер разработки запущен, и откройте страницу [**http://127.0.0.1:8000/**](http://127.0.0.1:8000/) , затем кликните на любой курс.

Если вы авторизованы, то увидите кнопку ENROLL NOW под описанием:



Рис. 11.4 Страница курса для авторизованных пользователей

Если же вы не авторизованы, то увидите кнопку REGISTER TO ENROLL.

**Доступ к содержимому курсов**

Теперь необходимо добавить страницу, показывающую студентам курсы, в которых они уже принимают участие, и страницу содержимого курса.

**Для этого отредактируйте файл views.py приложения students** и добавьте в него следующий код:

from django.views.generic.list import ListView

from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView): # page 357

model = Course

template\_name = 'students/course/list.html'

def get\_queryset(self): # page 357

qs = super(StudentCourseListView, self).get\_queryset()

return qs.filter(students\_\_in=[self.request.user])

Этот обработчик будет формировать список курсов, слушателем которых является студент. Мы используем примесь LoginRequiredMixin, чтобы только авторизованные пользователи могли иметь доступ к этой странице. Наш обработчик также наследуется от класса ListView, чтобы отображать объекты модели Course в виде списка. Чтобы получить только курсы, связанные с текущим пользователем, мы переопределили метод get\_queryset() и отфильтровали QuerySet курсов по связи ManyToManyField со студентом.

**Теперь добавьте в файл views.py** такой фрагмент:

from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView): # page 357

model = Course

358

template\_name = 'students/course/detail.html'

def get\_queryset(self): # page 358

qs = super(StudentCourseDetailView, self).get\_queryset()

return qs.filter(students\_\_in=[self.request.user])

def get\_context\_data(self, \*\*kwargs): # page 358

context = super(StudentCourseDetailView, self).get\_context\_data(\*\*kwargs)

# Получаем объект курса.

course = self.get\_object()

if 'module\_id' in self.kwargs:

# Получаем текущий модуль по параметрам запроса.

context['module'] = course.modules.get(id=self.kwargs['module\_id'])

else:

# Получаем первый модуль.

context['module'] = course.modules.all()[0]

return context

Это обработчик StudentCourseDetailView. Мы переопределили метод get\_queryset() , чтобы ограничить QuerySet курсов и работать только с теми, на которые записан текущий пользователь. Мы также переопределили метод get\_context\_data(), чтобы добавить в контекст шаблона данные о модуле, если его идентификатор был передан в параметре module\_id URLʼа. В противном случае мы показываем содержимое первого модуля. Так студенты смогут переходить от одного модуля курса к другому.

**Отредактируйте файл urls .py приложения students** и добавьте в него следую-

щие шаблоны URLʼов:

path('courses/', # page 358

views.StudentCourseListView.as\_view(),

name='student\_course\_list'),

path('course/<pk>/', # page 358

views.StudentCourseDetailView.as\_view(),

name='student\_course\_detail'),

path('course/<pk>/<module\_id>/', # page 358

views.StudentCourseDetailView.as\_view(),

name='student\_course\_detail\_module'),

Самое время добавить HTML-шаблоны.

**Для этого создайте в каталоге templates/students/ приложения students** следующую структуру файлов:

course/

detail.html

list.html

**Добавьте в шаблон students/course/list.html** приведенный ниже фрагмент

разметки:

{% extends "base.html" %}

{% block title %}My courses{% endblock %}

{% block content %}

<h1>My courses</h1>

359

<div class="module">

{% for course in object\_list %}

<div class="course-info">

<h3>{{ course.title }}</h3>

<p><a href="{% url "student\_course\_detail" course.id %}">

Access contents</a></p>

</div>

{% empty %}

<p>

You are not enrolled in any courses yet.

<a href="{% url "course\_list" %}">Browse courses</a>

to enroll in a course.

</p>

{% endfor %}

</div>

{% endblock %}

Этот шаблон мы будем использовать для списка курсов студента. Помните,

что после регистрации пользователь перенаправляется на URL с именем student\_course\_list.

Давайте будем **перенаправлять студентов на эту же страницу** после авторизации на сайте.

**Для реализации перенаправления добавьте в файл settings .py проекта educa**

следующие строки:

from django.urls import reverse\_lazy

LOGIN\_REDIRECT\_URL = reverse\_lazy('student\_course\_list') # page 359

Это настройка, которую использует пакет Django auth, для определения

адреса, куда перенаправлять пользователя после успешной авторизации на

сайте, если параметр next не задан явно. Теперь при входе в аккаунт студент

будет автоматически переходить на список своих курсов.

**Отредактируйте файл students/templates/students/course/detail.html и добавьте** такой фрагмент:

{% extends "base.html" %}

{% block title %}

{{ object.title }}

{% endblock %}

{% block content %}

<h1>

{{ module.title }}

</h1>

<div class="contents">

<h3>Modules</h3>

<ul id="modules">

{% for m in object.modules.all %}

<li data-id="{{ m.id }}" {% if m == module

%}class="selected"

{% endif %}>

<a href="{% url "student\_course\_detail\_module" object.id m.id %}">

<span>

Module <span class="order">{{ m.order|add:1 }}

</span>

360

<br>

{{ m.title }}

</a>

</li>

{% empty %}

<li>No modules yet.</li>

{% endfor %}

</ul>

</div>

<div class="module">

{% for content in module.contents.all %}

{% with item=content.item %}

<h2>{{ item.title }}</h2>

{{ item.render }}

{% endwith %}

{% endfor %}

</div>

{% endblock %}

Этот шаблон предназначен для отображения курса и его содержимого студентам, которые на него записались. Для начала мы генерируем список всех

модулей курса, выделяя текущий. Затем проходим по содержимому текущего

модуля и формируем соответствующий фрагмент разметки с помощью записи

вида {{ item.render }}. Метод render() для модели содержимого курса мы добавим чуть позже.

**Отображение различного типа содержимого**

На нашем сайте может быть несколько типов содержимого модуля: текст, картинки, видео или файлы. Нам необходимо по-разному отображать страницу

с содержимым курса в зависимости от типа. Давайте реализуем это.

courses/models.py

**Отредактируйте файл models.py приложения courses** и добавьте метод render() **в модель ItemBase**, как показано ниже:

from django.template.loader import render\_to\_string

from django.utils.safestring import mark\_safe

class ItemBase(models.Model): # page 360

# ...

def render(self): # page 360

return render\_to\_string('courses/content/{}.html'.format(

self.\_meta.model\_name), {'item': self})

В этом методе мы вызываем функцию render\_to\_string(), чтобы сгенерировать шаблон с контекстом и получить результат в виде строки. Каждый тип содержимого будет использовать соответствующий ему шаблон, полученный по названию модели. Чтобы динамически формировать имя шаблона, обратимся к атрибуту self.\_meta.model\_name модели. Метод render() предоставляет общий интерфейс для генерации шаблона под конкретный тип содержимого.

361

**Создайте в каталогe tеmplates/courses /приложения courses** следующую структуру файлов:

content/

text.html

file.html

image.html

video.html

**В файл courses/content/text.html вставьте** такую строку:

{{ item.content|linebreaks|safe }}

**Отредактируйте шаблон courses/content/file.html** и добавьте следующий код:

<p><a href="{{ item.file.url }}" class="button">Download file</a></p>

**Затем откройте файл courses/content/image.html** и вставьте в него строку:

<p><img src="{{ item.file.url }}"></p>

При использовании полей модели ImageField и FileField необходимо настроить проект в режиме разработки на работу с файлами, загружаемыми пользователями.

**Для этого откройте settings.py проекта educa** и добавьте две строки:

MEDIA\_URL = '/media/' # page 361

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media/') # page 361

Помните, что настройка MEDIA\_URL – это базовый URL, на основе которого будут формироваться ссылки к медиафайлам, расположенным в файловой системе в папке MEDIA\_ROOT.

Music\_training/urls.py

**Отредактируйте файл urls.py проекта** и добавьте **в начало файла** такой фрагмент:

from django.conf import settings

from django.conf.urls.static import static

Затем перейдите **на последнюю строку файла** и вставьте приведенный ниже

код:

if settings.DEBUG: # page 361

urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT) # page 361

Теперь проект готов к загрузке и отображению файлов пользователей. Сервер разработки Django возьмет на себя обязанность по поиску медиафайлов

в системе. Этот способ используется только при разработке (настрокаDEBUG

равна True).

**Никогда не применяйте такую конфигурацию в боевом режиме.** Об

этом мы узнаем в главе 13.

Осталось создать шаблон для отображения видео. Мы будем использовать

пакет Django-embed-video. Это стороннее приложение для Django, которое позволяет встраивать в шаблоны видео из таких источников, как YouTube, Vimeo. Пользователю будет достаточно указать адрес видео.

Установите пакет с помощью pip:

**pip install django-embed-video==1.1.2**

362

Подключите его в список установленных приложений в файле settings.py

проекта:

INSTALLED\_APPS = [

# ...

'embed\_video', # page 362

]

Полную документацию пакета django-embed-vid eмoожно найти на странице

https://django-embed-video.readthedocs.io/en/latest/.

**Отредактируйте файл courses/content/video.html и** вставьте такие строки:

{% load embed\_video\_tags %}

{% video item.url "small" %}

Теперь запустите сервер для разработки и перейдите в браузере на страницу

[**http://127.0.0.1:8000/course/mine/**](http://127.0.0.1:8000/course/mine/).

Войдите на сайт под именем пользователя, который принадлежит группе

«Владелец курса» (Instructors), и добавьте в модуль содержимое разных типов.

Чтобы создать видео, просто скопируйте его URL с сайта YouTube (например,

https://www.youtube.com/watch?v=bgV39DlmZ2U) и вставьте его url в поле формы.

После того как вы закончите редактировать курс, перейдите на главную

страницу по адресу **http://127.0.0.1:8000/**, кликните на курс и нажмите кнопку

ENROLL NOW. Вы будете зачислены на курс и перенаправлены по адресу с именем шаблона student\_course\_detail. Вы увидите содержимое курса такого вида:

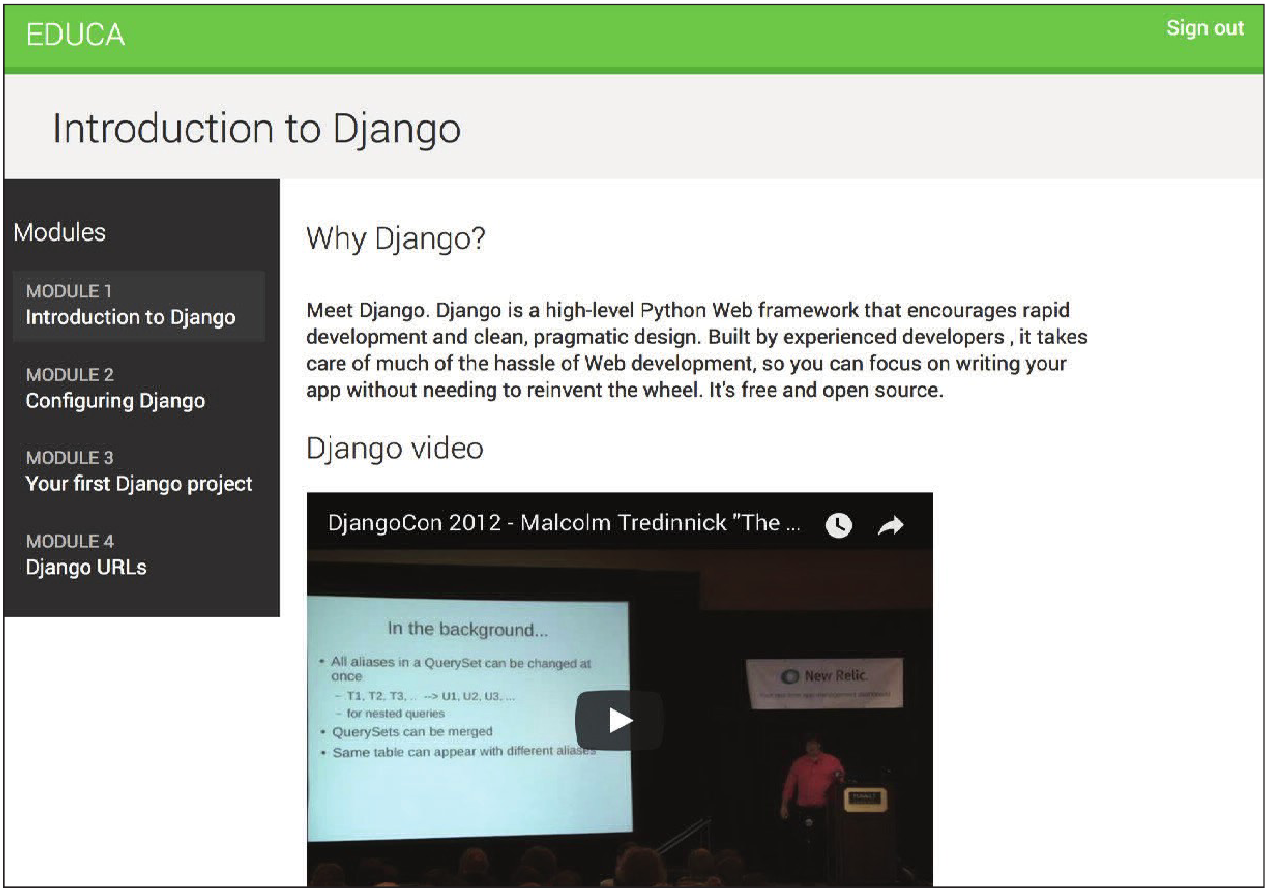


Рис. 11.5 Страница курса с несколькими типами содержимого

(текст и видео)

363

Поздравляем! Вы только что создали единый интерфейс для формирования

шаблонов содержимого разных типов.

**Использование фреймворка для кеширования**

Зачастую обработка запроса в веб-приложении подразумевает три этапа: об-

ращение к базе данных, какие-либо вычисления и преобразования результата

выборки и формирование ответа. Это гораздо более требовательный к произ-

водительности системы процесс, чем статичный сайт.

При этом для обработки некоторых запросов могут возникать задержки

при росте числа пользователей приложения. В таких случаях может помочь

кеширование. Вы можете кешировать в базу данных результаты запросов, ре-

зультаты каких-то вычислений, сгенерированный HTML-шаблон или любые

другие дорогостоящие операции. Так вам удастся снизить время отклика на

запрос.

Django предоставляет подсистему кеширования, которая позволит опреде-

лить, какие фрагменты кода и результаты вычислений должны быть кеширова-

ны. Например, это могут быть отдельные запросы, результат работы конкрет-

ного обработчика, часть или весь HTML-шаблон или даже весь сайт. Элементы

хранятся в подсистеме кеширования в течение определенного срока, который

вы можете переопределить.

Вот каким образом обычно работает подсистема кеширования при обработ-

ке HTTP-запроса:

1) пытается найти запрашиваемые данные в кеше;

2) если это удалось, возвращает ответ;

3) если данные не нашлись, выполняет такие шаги:

а) делает запрос или вычисления в соответствии с логикой обработчика;

б) сохраняет результат в кеш;

в) возвращает данные.

Более подробно про работу подсистемы кеширования Django можно про-

честь на странице документации https://docs.djangoproject.com/en/2.0/topics/

cache/.

Доступные бэкэнды кеширования

В Django реализовано несколько бэкэндов кеширования. Вот их краткое опи-

сание:

* backends.memcached.Memcached или Cache,backends.memcached.PyLibMCCache, –

бэкэнды для Memcached. Memcached. Это быстрая и эффективная систе-

ма кеширования, работающая с оперативной памятью. Какой именно из

классов использовать, зависит от того, как вы настроите взаимодействие

Memcached со своим Python-кодом;

* backends.db.DatabaseCache – использует в качестве хранилища кешей базу данных;

364

* backends.lebased.FileBasedCache – сохраняет результаты в файловую си-

стему, сериализует и хранит каждое кешированное значение в отдель-

ном файле;

* backends.locmem.LocMemCache – бэкэнд для кеширования в памяти. Используется по умолчанию;
* backends.dummy.DummyCache – фиктивный бэкэнд кеширования, применяе-

мый только при разработке. Он реализует интерфейс бэкэнда кеширо-

вания, но фактически не сохраняет никакие результаты.

Для хорошей производительности системы обычно используют системы кеширования, основанные на оперативной памяти, например Memchached.

Установка Memcached

В проекте мы будем использовать Memcached. Это приложение при запуске по-

лучает от операционной системы заданный в настройках объем оперативной

памяти для кеширования. Как только в хранилище становится недостаточно

места для новых значений, Memcached перезаписывает старые.

Скачайте приложение с сайта: <https://memcached.org/downloads>.

Если вы работаете с операционной системой Linux, для установки достаточно выполнить команду:

./configure && make && make test && sudo make install

Если вы работаете с macOS X, можете установить Memcached через менед-

жер пакетов Homebrew с помощью комаbнrдeыw i nstall memcach. eЧdтобы скачать Homebrew, перейдите на сайт

https://brew.sh/.

После установки Memcached откройте консоль и выполните команду:

memcached -l 127.0.0.1:11211

По умолчанию приложение будет запущено на 11211 порту. Но вы можете задать любой другой с помощью флага – l.

Более подробную информацию о Memchached можно найти в официальной документации на странице <https://memcached.org/>.

После установки приложения необходимо установить также и Python-пакет

для взаимодействия с Memcached из Python-кода.

Выполните такую команду:

pip install python-memcached==1.59

Настройки кеширования

Django поддерживает несколько настроек для конфигурации бэкэнда кеширо-

вания:

* CACHES – словарь используемых в проекте систем кеширования;
* CACHE\_MIDDLEWARE\_ALIAS – псевдонимы кешей;
* CACHE\_MIDDLEWARE\_KEY\_PREFIX – префиксы для ключей кешей.
* Когда ваш проект работает с несколькими сайтами, префиксы позволяют избежать коллизий имен ключей;

365

* CACHE\_MIDDLEWARE\_SECONDS – продолжительность хранения кешированных страниц в секундах.

Подсистему кеширования проекта конфигурируют с помощью настройки

CACHES. Это словарь, который задает параметры конфигурации для каждого бэ-

кэнда, например такие:

* BACKEND – используемый класс для бэкэнда;
* KEY\_FUNCTION – функция для получения ключа кеша. Она принимает в качестве аргументов префикс, версию и некоторый начальный ключ в виде строки. Затем преобразует их в ключ, используемый для кеширования;
* KEY\_PREFIX – префикс для всех ключей бэкэнда;
* LOCATION – расположение результата кеширования. В зависимости от ис-

пользуемого класса бэкэнда эта настройка может принимать значения

в виде пути в файловой системе, хоста и порта или имени для бэкэндов

на основе оперативной памяти;

* OPTIONS – любые дополнительные параметры, которые может принимать конкретный класс бэкэнда;
* TIMEOUT – время хранения результатов кеширования в секундах. По умолчанию оно равно 300, т. е. 5 мин. Если установить эту настройку равной None, срок хранения данных не будет ограничен;
* VERSION – номер версии кешированных данных. Нужно для добавления версий при кешировании.

Добавление Memcached в проект

Давайте настроим проект для взаимодействия с Memcached.

**Откройте файл settings.py проекта educa** и вставьте в него такой фрагмент:

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',

'LOCATION': '127.0.0.1:11211',

}

}

Мы указали в качестве бэкэнда класc MeсmcahedCache. Адрес, по которому

Django сможет к нему обратиться, задали в видe addrеss:port. При этом, если

вы используете несколько рабочих процессов Memcached, в настройке LOCATION необходимо задать список их адресов.

Мониторинг Memcached

Чтобы анализировать работу Memcached, воспользуемся сторонней библио-

текой django-memcache-status.Это приложение собирает статистику по каждому рабочему процессу Memcached и отображает ее на сайте администрирования.

Установите пакет с помощью команды:

pip install django-memcache-status==1.3

366

Отредактируйте фаsйeлt tings.py проекта и добавьте приложение 'memcache\_status' в настройку INSTALLED\_APPS:

INSTALLED\_APPS = [

# ...

'memcache\_status',

]

Убедитесь, что Memcached и сервер для разработки запущены, после чего

откройте в браузере страниhцttуp ://127.0.0.1:8000/admin./

Войдите под логином администратора системы. Вы увидите такой блок:

Рис. 11.6 Информация об использовании памяти Memcached

Этот график демонстрирует статистику использования Memcached. Зеленый

цвет соответствует количеству свободного пространства, а красный – занятого

кешированными данными. Если вы кликнете на заголовок блока, то увидите

детализированную статистику рабочего процесса Memcached.

Мы добавили систему кеширования Memcached и инструмент для анализа

ее работы. Самое время реализовать кеширование на сайте!

Уровни кеширования

Django поддерживает кеширование данных на нескольких уровнях, представ-

ленных ниже:

* низкоуровневый API – предоставляет возможность кешировать наи-

меньшую единицу вычислений (запросы или вычисления);

* уровень обработчиков– кешируются результатыо бработки одного

HTTP-запроса;

* уровень шаблонов – применяется для добавления в кеш результата гене-

рации HTML-шаблона или его фрагмента;

* уровень сайта – кеширует весь сайт.

Хорошо обдумайте стратегии кеширования, перед тем как применять их в проекте. Для начала необходимо оптимизировать наиболее дорогие с точки зрения вычислений запросы и операции.

**Использование низкоуровнего API кеширования**

Этот способ позволяет сохранять в кеше данные любого размера. Все необхо-

димые классы и функции реализованы в пакете django.core.cache.

Чтобы начать использовать низкоуровневый кеш Django, достаточно импортировать его:

from django.core.cache import cache

367

Так вы получите доступ к кешу по умолчанию. Запись эквивалентна записи

caches['default'] . Чтобы обратиться к другому кешу, описанному в настройках

приложения, используйте следующие строки:

from django.core.cache import caches

my\_cache = caches['alias']

Давайте посмотрим, как работает низкоуровневый API кеширования. Откройте консоль и

выполните команду python manage.py shell,

после чего введите такие строки:

>>> from django.core.cache import cache

>>> cache.set('musician', 'Django Reinhardt', 20)

Мы обратились к кешу по умолчанию и установили значение 'Django Reinhardt' по ключу 'musician' на 20 с. с помощью записи set (key, value, timeout).

Если не задать третий аргумент метода, Django использует значение по умолчанию из настройки CACHES.

Теперь выполните такой код:

>>> cache.get('musician')

'Django Reinhardt'

Мы получили значение, которое ранее добавили в кеш. Подождите 20 с и по-

пытайтесь снова обратиться к кешу по этому ключу:

>>> cache.get('musician')

Вы не увидите результата, т. к. ключ 'musician' был удален из кеша по истечении срока действия.

Избегайте установки неограниченного срока хранения данных в кеше, так вы обезопасите себя от получения неактуальной информации.

Давайте попробуем кешировать результат выполнения QuerySetʼа:

>>> from courses.models import Subject

>>> subjects = Subject.objects.all()

>>> cache.set('all\_subjects', subjects)

Мы обратились к QuerySetʼу модели Subject и сохранили результат выборки

в кеше по ключу 'all\_subjects' . Попробуйте получить его из кеша вызовом метода get():

>>> cache.get('all\_subjects')

<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,

<Subject: Programming>]>

Мы можем применить кеширование QuerySetʼов в некоторых наших обработчиках. Отредактируйте фvаiйeлw s.py приложенияc ourses и добавьте такой импорт:

from django.core.cache import cache

368

В методе get() обработчика CourseListView найдите строку

subjects = Subject.objects.annotate(total\_courses=Count('courses'))

и замените ее следующим образом:

subjects = cache.get('all\_subjects')

if not subjects:

subjects = Subject.objects.annotate(

total\_courses=Count('courses'))

cache.set('all\_subjects', subjects)

В этом коде мы сначала пытаемся получить данные из кеша по ключу all\_students. Если метод cache.get() возвращает None, следовательно, QuerySet не находится в кеше (например, еще не был туда добавлен или истек срок его хранения). В этом случае выполняем запрос в базу данных, чтобы получить объекты модели Subject и количество курсов для каждого из них, после чего сохраняем результат запроса в кеш.

Запустите сервер для разработки и откройте в браузере страницу http://127.0.0.1:8000/. Когда обработчик выполняется первый раз, кеш пуст, осуществляется запрос в базу данных.

Откройте страницу http://127.0.0.1:8000/admin / и разверните блок статистики Memcached.

Вы увидите что-то похожее:

Рис. 11.7 Статистика использования Memcached

Обратите внимание на поле CurrItems, его значение должно быть равно 1.

Это говорит нам о том, что сейчас в кеше находится только один объект. Поле

Get Hits показывает, сколько запросов в кеш было произведено успешно, а Get

Misses – наоборот, количество неудачных обращений к кешу. Поле Miss Ratio

выводит результирующую оценку по этим двум параметрам.

369

Теперь перейдите обратно на главную страницу http:// 127.0.0.1:8000/ и пе-

резагрузите ее несколько раз. Теперь в статистике использования кеша вы увидите обновленные данные с увеличенным количеством успешных запросов Get Hits и Cmd Get.

Кеширование на основе динамических данных

Часто на практике появляется необходимость кешировать динамические данные, которые формируются в зависимости от каких-то входных параметров.

В таких случаях необходимо формировать уникальные ключи кеширования.

Отредактируйте файл views.py приложения courses и измените обработчик

CourseListView, чтобы он выглядел следующим образом:

class CourseListView(TemplateResponseMixin, View):

model = Course

template\_name = 'courses/course/list.html'

def get(self, request, subject=None):

subjects = cache.get('all\_subjects')

if not subjects:

subjects = Subject.objects.annotate(total\_courses=Count('courses'))

cache.set('all\_subjects', subjects)

all\_courses = Course.objects.annotate(total\_modules=Count('modules'))

if subject:

subject = get\_object\_or\_404(Subject, slug=subject)

key = 'subject\_{}\_courses'.format(subject.id)

courses = cache.get(key)

if not courses:

courses = all\_courses.filter(subject=subject)

cache.set(key, courses)

else:

courses = cache.get('all\_courses')

if not courses:

courses = all\_courses

cache.set('all\_courses', courses)

return self.render\_to\_response({'subjects': subjects,

'subject': subject,

'courses': courses})

Мы добавили кеширование курсов, отфильтрованных по конкретному предмету. Для первого случая мы используем ключ all\_courses. Если в запросе был передан ID предмета, создаем ключ кеширования из строки 'subject\_{}\_courses'.format(subject.id) .

Важно понимать, что когда вы получаете QuerySet из кеша, то нельзя получить на его основе новый, т. е. такая запись не будет работать:

courses = cache.get('all\_courses')

courses.filter(subject=subject)

Вместо этого вам нужно создать новый объект QueryCoursSeet а ʼobjects.annotate(total\_modules=Count('modules',)) который не будет выполняться до непосредственного обращения к нему. Затем, если соответствующее значение

370

не было найдено в кеше, следует применить к нему фильтрацию, например all\_courses.

371

Кеширование результатов работы обработчиков

Для добавления в кеш результатов обработки HTTP-запросов в пакете djagno.views.decorators.cache реализован декоратор cache\_page. При его использовании необходимо передать обязательный параметр timeout – срок жизни кешированного значения в секундах.

Давайте применим его в проекте. Отредактируйте файл urls. py приложения

students и добавьте такой импорт:

from django.views.decorators.cache import cache\_page

Затем оберните обработчики student\_course\_detail и student\_course\_detail\_module в декораторe cache\_pag, как показано ниже:

path('course/<pk>/',

cache\_page(60 \* 15)(views.StudentCourseDetailView.as\_vie)w,()

name='student\_course\_detail'),

path('course/<pk>/<module\_id>/',

cache\_page(60 \* 15)(views.StudentCourseDetailView.as\_vie)w,()

name='student\_course\_detail\_module'),

Т

еперь результаты обработчика StudentCourseDetailView будут кешироваться

на 15 мин.

При кешировании обработчиков в качестве ключа используется имя соответствующих шаблонов URLʼов. Поэтому если несколько шаблонов ссылаются на один и тот же обработчик, они будут кешированы с разными именами.

Использование кеширования для сайта

Этот способ самый высокоуровневый, он позволяет реализовать кеширование

всего сайта.

Чтобы проект мог работать с кешированием сайтов, отредактируйте файл

settings.py и добавьте классы UpdateCacheMiddleware и UpdateCacheMiddleware в настройку MIDDLEWARE:

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.cache.UpdateCacheMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.cache.FetchFromCacheMiddleware',

# ...

]

Помните, что промежуточные слои при обработке запроса выполняются

в том порядке, в каком указаны в настройке, а при возврате ответа – в противо-

положном. Поэтому необходимо разместить класс UpdateCacheMiddleware перед CommonMiddlewаre,класс FetchFromCacheMiddleware – после.

Теперь добавьте эти строки в фseаtйtiлn gs.py проекта:

372

CACHE\_MIDDLEWARE\_ALIAS = 'default'

CACHE\_MIDDLEWARE\_SECONDS = 60 \* 15 # 15 minutes

CACHE\_MIDDLEWARE\_KEY\_PREFIX = 'educa'

С помощью таких настроек мы задаем псевдоним кеша и время жизни ке-

шированных данных в 15 мин по умолчанию. Мы также указали префикс для

всех ключей, чтобы избежать их пересечения при использовании одного рабо-

чего процесса Memcached с несколькими проектами. Теперь наш сайт будет ке-

широван и станет возвращать одно и то же содержимое GдEлT-яз авсперхо сов.

Мы добавили кеширование на уровне сайта только для того, чтобы проде-

монстрировать возможности этого способа. На самом деле это не подходит

для платформы онлайн-обучения, т. к. курсы могут меняться, а пользователям

необходимо показывать актуальную информацию. В нашем случае наиболее

подходящий вариант – кеширование на уровне обработчиков и шаблонов, ко-

торые используются при просмотре студентами курсов.

Мы рассмотрелис пособы кеширования,к оторые поддерживаетD jango.

В реальных проектах стоит относиться к кешированию с умом и тщательно

продумывать стратегии кеширования (начинать с наиболее затратных опера-

ций, а только после этого переходить к наименее критичным частям проекта).

резюме

В этой главе мы создали страницы для доступа к курсам и реализовали регист-

рацию пользователей и запись на курсы, а также подключили к проекту при-

ложение Memcached и рассмотрели несколько уровней кеширования.

В следующей главе мы реализуем RESTful API для проекта.

**Глава12**

**Реализация API**

В предыдущей главе вы узнали, как добавить в проект возможность регистра-

ции студентов и записи на курсы, реализовать обработчики для страниц кур-

сов и подключить к Django-проекту систему кеширования. В этой главе мы рас-

смотрим следующие темы:

* создание RESTful API;
* аутентификация и ограничение доступа к API-обработчикам;
* создание наборов обработчиков и маршрутизаторов.

создание Restful API

Иногда в проектах появляется необходимость взаимодействия с другими системами для отображения данных вашего веб-приложения. Для этих целей реализуют специальный интерфейс, API, который определяет точки взаимодействия двух систем.

Существует несколько способов организации API, но наиболее часто при-

меняют REST. REST – это аббревиатура от термина передача состояния управ-

ления (Representational State Transfer). Такой способ взаимодействия двух систем основан на ресурсах. Модель приложений представляет собой ресурсы, а HTTP-заголовки – действия. Например GET, POST, PUT и DELETE используются для получения, изменения, создания или удаления объектов. В этом подходе также активно применяют коды HTTP-ответов (например, статус 200-ыx означают успешную операцию, 400 – ошибки при обработке запроса и т. д.).

Общепринятые форматы взаимодействия RESTful – JSON и XML. Мы реализуем в проекте поддержку REST API посредством сериализации объектов в JSON. Наш интерфейс взаимодействия будет предоставлять такие точки доступа, как:

* список предметов;
* список всех курсов;
* содержимое курсов;
* запись на курс.

Мы можем реализовать API с помощью обработчиков Django. Однако существует несколько сторонних приложений,к оторые упрощают создание

374

REST API. Наиболее часто для этих целей применяют фреймворк Django REST

Framework.

**установка Django restframework**

Фреймворк Django REST позволяет вам легко добавить REST API к Django-проекту. Полную документацию по этому приложению можно найти на странице https://www.django-rest-framework.org/.

Откройте консоль и установите его с помощью pip:

pip install djangorestframework==3.8.2

Отредактируйте файл settings.py проекта edu a и добавьте rest\_framework

в список INSTALLED\_APPS для активации приложения:

INSTALLED\_APPS = [

# ...

'rest\_framework',

]

Затем допишите такую настройку:

REST\_FRAMEWORK = {

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'

]

}

С помощью настройки REST\_FRAMEWORK можно дополнительно сконфигурировать фреймворк Django REST, который поддерживает множество различ-ных параметров для переопределения поведения по умолчанию. Например, настройка DEFAULT\_PERMISSION\_CLA SоSтEвSечает за разрешения по умолчанию для доступа к чтению, созданию, изменению и удалению объектов. Мы указали класс DjangoModelPermissionsOrAnonRead,Only поэтому анонимные пользователи смогут только просматривать, но не изменять данные, а авторизованные будут иметь доступ ко всем четырем действиям. Более подробно разграничение доступа мы разберем чуть позже в секции «Обработка аутентификации пользователей».

Полный список доступных настроек фреймворка приведен на странице

https://www.django-rest-framework.org/api-guide/settings/.

**Определение сериализаторов**

После установки Django REST нам необходимо задать способ, с помощью которого данные моделей будут преобразованы в JSON и обратно. Фреймворк предоставляет несколько базовых классов сериализаторов, которые работают с объектами:

* Serializer – сериализует обычные Python-классы;
* ModelSerializer – преобразует объекты моделей Django;Установка Django REST Framewor k 375
* HyperlinkedModelSerializer – аналогичен классу ModelSerializer, но для связанных объектов формирует http-ссылки, а не их внешние ключи.

Давайте создадим наш первый сериализатор. Добавьте в каталог приложения courses такую структуру файлов:

api/

\_\_init\_\_.py

serializers.py

Мы будем описывать всю функциональность, касающуюся API, в пaпке аpi,

чтобы код был структурирован.

**Отредактируйте файл serializers.py** и добавьте в него следующий код:

from rest\_framework import serializers

from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):

class Meta:

model = Subject

376

Давайте посмотрим, как преобразовать входные параметры в подходящие

объекты. Выполните в Python-консоли следующие команды:

>>> from io import BytesIO

>>> from rest\_framework.parsers import JSONParser

>>> data = b'{"id":4,"title":"Programming","slug":"programming"}'

>>> JSONParser().parse(BytesIO(data))

{'id': 4, 'title': 'Programming', 'slug': 'programming'}

Когда у вас есть строка в формате JSON, для получения соответствующего

Python-объекта используйте класс JSON Parser.

Фреймворк Django REST также предоставляет класс Renderer, который позволяет формировать ответы на запросы к API. Для определения конкретного

класса рендерера фреймворк смотрит на тип объекта, который нужно преобразовать. Тип ответа зависит от значения HTTP-заголовка запрса Acceоpt. Дополнительно можно настроить формат ответа по префиксам в URLʼах запросов.

Например, для формирования JSON-ответа будет задействован класс JSONRenderer.

Вернитесь к консоли и выполните следующий код, чтобы сформировать ответ из объектsаe rializer предыдущего примера:

>>> from rest\_framework.renderers import JSONRenderer

>>> JSONRenderer().render(serializer.data)

Вы увидите такой вывод:

b'{"id":4,"title":"Programming","slug":"programming"}'

Мы применили класс JSONRenderer, чтобы сформировать JSON-ответ. По

умолчанию Django REST использует два рендерера: JSONRenderer и BrowsableAPIRenderer. Второй предоставляет веб-интерфейс для просмотра API. Вы можете указать необходимый рендерер по умолчанию с помощью атрибута DEFAULT\_RENDERER\_CLASS в настройке ESREST\_FRAMEWORK.

Более подробную информацию о рендерерах и парсерах можно найти на

страницах https://www.django-rest-framework.org/api-guide/rendи erhetrtsp/s ://

www.django-rest-framework.org/ api-guide/parsers/ соответственно.

Создание обработчиков списка и подробностей

Во фреймворке Django REST реализован базовый набор классов и примесей,

которые помогают разработчику создавать обработчики запросов к API. Они

могут использоваться для точек доступа к функциям получения, изменения,

создания или удаления объектов. Список всех базовых классов и примесей

можно найти в официальной документации на странице http://www.django-

rest-framework.org/api-guide/generic-views/.

Давайте создадим обработчики списка и подробностей предметов – объек-

тов моделиS ubject.

**Добавьте новый файл views.py в папке courses/api/** и вставьте в него такой код:

377

from rest\_framework import generics

from ..models import Subject

from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):

queryset = Subject.objects.all()

serializer\_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):

queryset = Subject.objects.all()

serializer\_class = SubjectSerializer

В этом коде мы используем базовые клLаiсstсAыP IView и RetrieveAPIVie.w Мы

добавим значение идентификатора предмета, pпkо, лве качестве параметра

в URL, чтобы получить информацию по конкретному предмету. У обоих обработчиков мы определили два атрибута:

* queryse t– начальный QuerySet для получения объектов;
* serializer\_class – класс для сериализации объектов.

Давайте добавим шаблоны URLʼов для этих обработчиков.

**Создайте новый файл urls.py в папке courses/api/** и добавьте в него следующий фрагмент:

from django.urls import path

from . import views

app\_name = 'courses'

urlpatterns = [

path('subjects/',

views.SubjectListView.as\_view(),

name='subject\_list'),

path('subjects/<pk>/',

views.SubjectDetailView.as\_view(),

name='subject\_detail'),

]

**Откройте файлr uls.py проекта educa и** подключите шаблоны, которые мы

только что определили:

urlpatterns = [

# ...

path('api/', include('courses.api.urls', namespace='api')),

]

Мы добавили параметр namespace, чтобы определить пространство имен

для URLʼов внешнего API. Убедитесь, что сервер запущен командой python

manage.p ryunserver.

Откройте консоль и обратитесь к приложению по адресу

http://127.0.0.1:8000/api/subjects/ с помощью утилитcы url:

curl http://127.0.0.1:8000/api/subjects/

Вы увидите ответ, похожий на этот:

[

{"id":1,"title":"Mathematics","slug":"mathematics"},378  Реализация API

{"id":2,"title":"Music","slug":"music"},

{"id":3,"title":"Physics","slug":"physics"},

{"id":4,"title":"Programming","slug":"programming"}

]

HTTP-ответ содержит список всех предметов, которые созданы в вашем приложении, в формате JSON. Если ваша операционная система не поддерживает утиліту curl по умолчанию, скачайте ее с сайта https://curl.haxx.se/dlwiz/. Кроме этого, вместcо url можно использовать любой другой инструмент по отправке HTTP-запросов, например расширение для браузеров Postman. Для его скачивания обратитесь к сайту https://www. getpostman.com/.

Откройте в браузере страницу http://127.0.0.1:8000/api/subjects/ . Вы увидите описание API, которое формирует фреймворк Django REST:

Рис. 12.1 Описание API в интерфейсе Django REST Framework

Такие страницы генерируются благодаря рендереру BrowsableAPIRenderer.

Он показывает заголовки и содержимое HTTP-ответа на запрос. Аналогич-

ным образом вы можете посмотреть результат выполнения запроса подроб-

379

ностей о предмете, если добавите его ID в URL.

Например, откройте страницу

http://127.0.0.1:8000/api/subjects/1/ . Вы увидите объект моделиmSubject, преобразованный в формат JSON.

**Создание вложенных сериализаторов**

Мы собираемся реализовать сериализатор для модели Course.

**Отредактируйте файл api/serializers.py приложения courses** и добавьте следующий фрагмент:

from ..models import Course

class CourseSerializer(serializers.ModelSerializer):

class Meta:

model = Course

380

Мы создали новый класс ModuleSerializer для сериализации модели Module.

Затем добавили его в атрибут Modules сериализатора курсов CourseSerializer,

указав, что он является вложенным. В качестве параметра передали many=Tru, e

чтобы обозначить, что для одного курса может быть множество модулей. Параметр read\_only сообщает фреймворку, что данные вложенного сериализатора не являются доступными для редактирования.

Откройте консоль и создайте новый объект класса CourseSerializer. Сгенерируйте данные курсов с помощью класса JSON Renderer, как мы делали это выше.

На этот раз информация в теге modules будет в виде не списка идентификаторов, а вложенного объекта, описывающего модули курса:

"modules": [

{

"order": 0,

"title": "Introduction to overview",

"description": "A brief overview about the Web Framework."

},

{

"order": 1,

"title": "Con”

381

Обработчик CourseEnrollView зачисляет студентов на курсы. В представленном выше фрагменте мы выполнили следующие действия:

1) создали обработчик, который наследуется от базовогоA PкIлViаeсw;са

2) определили метод post() для обработки POST-запросов. Другие методы не

будут обрабатываться;

3) для получения курса, на который происходит зачисление, обратились

к POST-параметру запроса pk. Если соответствующего курса не существует, будет сгенерировано исключение, вследствие чего пользователь получит ответ с кодом 404;

4) если удалось определить курс, создали связь между студентом и объек-том модели Course, после чего возвратили ответ с успешным статусом.

**Отредактируйте файл api/urls.py и** добавьте шаблон URLʼа для обработчика

CourseEnrollView:

path('courses/<pk>/enroll/',

views.CourseEnrollView.as\_view(),

name='course\_enroll'),

Сейчас обработчик готов и будет пытаться зачислить всех пользователей,

которые обратятся к нему. Но это не совсем корректно, т. к. даже неавторизованные пользователи имеют к нему доступ. Давайте посмотрим, как Django REST работает с разрешениями и ограничивает доступ.

**Обработка аутентификации пользователей**

Фреймворк Django REST предоставляет классы для управления аутентифи-

кацией. Если пользователь аутентифицирован, в объект запроса добавляется

атрибутr equest.user с объектом модели User. В противном случае добавляется

объект AnonymousUser.

Django REST Framework предоставляет такие бэкэнды аутентификации:

* BasicAuthentication – бэкэнд для базовой аутентификациип о HTTP.

Пользователь и пароль отправляются в зашифрованном виде в HTTP-

заголовкеA uthorization. Подробное описание этого механизма приведе-

но на странице https://en.wikipedia.org/wiki/Basic\_access\_authentication;

* TokenAuthentication – бэкэнд для авторизации по токену. В этом случае используется модель Token для сохранения токенов пользователей, которые добавляются к запросам в HTTP-заголовке Authorization;
* SessionAuthenticatio n– бэкэнд на основе сессий Django. Такой способ может быть полезен, когда браузерный код отправляет много AJAX-запросов к API сайта;
* RemoteUserAuthentication – делегирует процесс авторизации пользователя веб-серверу, который устанавливает переменную окружения REMOTE\_US.ER

Вы можете реализовать собственный бэкэнд. Для этого достаточно создать класс-наследник оBтa seAuthentication и описать для него метод authenticate() .

382

В Django REST предусмотрена возможность использования аутентификации

на уровне обработчиков или глобально, на весь проект. Управлять этим поведением можно с помощью настройки DEFAULT\_AUTHENTICATION\_CLASSES.

Аутентификация только определяет конкретного пользователя, выполняющего запрос, но не ограничивает доступ к обработчикам по ролям. Чтобы скрыть какие-либо обработчики от части пользователей, используйте разрешения.

Более подробную информацию об аутентификации во фреймворке Django

REST можно найти на странице https://www.django-restframework.org/api-guide/

authentication/.

Давайте добавим к нашему обработчику класс BasicAuthentication.

**Откройте файл api/views.py приложения courses** и задайте атрибут authentication\_classes обработчику CourseEnrollView таким образом:

from rest\_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):

authentication\_classes = (BasicAuthentication,)

# ...

Теперь мы будем определять пользователя по HTTP-заголовку Authorization

запроса.

**Ограничение доступа к обработчикам с помощью разрешений**

В Django REST реализована подсистема управления доступом пользователей

к сайту по аналогии с тем, как это сделано в Django. Вот, например, несколько

разрешений, определенных во фреймворке:

* AllowAn y– доступ разрешен всем;
* IsAuthenticated – доступ имеют только авторизованные пользователи;
* IsAuthenticatedOrReadOnly – доступ имеют все, но анонимные пользователи не могут изменять данные;
* IsAuthenticatedOrReadOnly - разрешение на основе django.contrib.auth.

У обработчика с таким уровнем доступа обязательно должен быть задан

атрибутq ueryset. Он будет обрабатывать запросы пользователей, которые имеют разрешение на обращение к указанной модели;

* DjangoObjectPermissions – разрешения Django по отношению к конкретным объектам.

Если доступ запрещен, автоматически формируется ответ с одним из таких

HTTP-статусов:

* HTTP 401 – пользователь не авторизован;
* HTTP 403 – доступ запрещен.

Подробная информация о работе с разрешениями в Django REST Framework

с примерами приведена на странице https://www.django-restframework.org/api-guide/permissions/.

**Отредактируйте фаaйpлi /views.py приложения courses** и добавьте атрибут

permission\_classes в обработчик CourseEnrollView:

383

from rest\_framework.authentication import BasicAuthentication

from rest\_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):

authentication\_classes = (BasicAuthentication,)

permission\_classes = (IsAuthenticated,)

# ...

Мы добавили разрешение IsAuthenticated. Теперь анонимные пользователи

не смогут получить доступ к обработчику. Давайте проверим, действительно

ли это так, отправив POST-запрос к API.

Убедитесь, что сервер для разработки запущен. Откройте другой сеанс

в консоли и выполните команду:

curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/

Вы увидите такой вывод:

HTTP/1.1 401 Unauthorized

...

{"detail": "Authentication credentials were not provided."}

Вы получили ошибку 401, потому что выполнили запрос как анонимный

пользователь. Давайте укажем данные студента, зарегистрированного в си-

стеме, и проверим, как работает ограничение доступа. Выполните следующую

команду, замениsвt udent:passwor dна актуальные логин и пароль вашего пользователя:

curl -i -X POST -u student:password http://127.0.0.1:8000/api/courses/1/enroll/

На этот раз вы увидите вывод с успешным статусом ответа:

HTTP/1.1 200 OK

...

{"enrolled": true}

Если перейти на сайт администрирования, то можно легко убедиться, что студент действительно зачислен на курс.

**Создание блоков обработчиков и их маршрутизаторов**

Класс ViewSets позволяет вам реализовать динамический API с несколькими

обработчикам и URLʼами, формируемыми с помощью объектов Router типа .

Используя набор, вы избегаете повторения кода. Такой набор содержит обработчики типичных действий (создания, получения, изменения и удаления

объектов) с помощью таких методовl,i s tк()ак, create(), retrieve() , update(),

partial\_update() и destroy().

Давайте создадим набор обработчиков для модели Course.

**Вставьте в файл api/views.py следующий фрагмент:**

from rest\_framework import viewsets

from .serializers import CourseSerializer

384

class CourseViewSet(viewsets.ReadOnlyModelViewSet):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

Мы унаследовали класс от ReadOnlyModelViewSet, который реализует доступ

только для чтения через метоreды trieve() и list() . Первый из них возвращает единственный объект, а второй – список.

**Отредактируйте файл api/urls.py** и добавьте маршрутизатор к набору обработчиков, который мы только что создали:

from django.urls import path, include

from rest\_framework import routers

from . import views

router = routers.DefaultRouter()

router.register('courses', views.CourseViewSet)

urlpatterns = [

# ...

path('', include(router.urls)),

]

Мы создали объект DefaultRouter и зарегистрировали набор обработчиков

с префиксом courses. Маршутизатор автоматически сформирует необходимые

URLʼы и будет передавать запрос в подходящий обработчик.

Откройте в браузере страницу http ://127.0.0.1:8000/api/ .

Вы увидите все об-работчики набора, соответствующие маршрутизDаeтfоaрulуt Router:

Рис. 12.2  Маршрутизатор и соответствующие обработчики запросов

Вы можете сделать запрос по адресу http://127.0.0.1:8000/api/courses/, чтобы

получить список всех курсов.

Полная документация по набору обработчиков приведена на сайте https://

www.django-rest-framework.org/apiguide/viewsets/. Подробнее о работе с маршру-тизаторами Django REST можно прочитать на странице https://www.django-rest-framework.org/api-guide/routers/.

385

Добавление собственных обработчиков в набор

Вы можете задать новые обработчики в наборе. Давайте добавим класс -CourseEnrollView и будем использовать его как собственный обработчик создания объекта.

**Отредактируйте файл аpi/views.py, чтобы кл**асс CourseViewSet выглядел

таким образом:

from rest\_framework.decorators import detail\_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

@detail\_route(methods=['post'],

authentication\_classes=[BasicAuthentication],

permission\_classes=[IsAuthenticated])

def enroll(self, request, \*args, \*\*kwargs):

course = self.get\_object()

course.students.add(request.user)

return Response({'enrolled': True})

Мы добавили новый метeод nroll(), который будет обрабатывать нестандартное действие – зачисление студентов на курсы. В этом фрагменте кода мы выполняем следующие шаги:

1) используем декоратор detail\_route, чтобы указать, что метод работает

с одним объектом, а не списком;

2) в качестве аргумента передаем в декоратор список HTTP-методов, с ко-

торыми может работать функція enroll() , а также классы для авторизации пользователей и проверки прав доступа;

3) вызываем метоsдe lf.get\_object() , чтобы получить объект Course;

4) добавляем свіязь students текущего пользователя и курса, на который он

пытается записаться.

Отредактируйте файл api/urls.py и удалите этот шаблон URLʼа, т. к. теперь он

будет генерироваться автоматически посредством маршрутизатора:

path('courses/<pk>/enroll/',

views.CourseEnrollView.as\_view(),

name='course\_enroll'),

Удалите также из файла api/ views.py класс CourseEnrollView.

Теперь передача запроса и обработка зачисления студента на курс полностью реализованы с помощью маршрутизатора и набора обработчиков. Адрес,

по которому пользователь может записаться на курс, формируется динамиче-

ски на основе имени метода nroll .

**Создание собственных разрешений**

Давайте добавим ограничение доступа к содержимому курсов для пользователей, не записанных на него. Только те, кто зачислен на курс, смогут просматривать модули и содержимое. Самый простой и надежный способ – создать

386

собственное разрешение. Django предоставляет базовыйB кasлeаPсeсr missio.n

Нас интересуют два его метода:

* has\_permission() – выполняет проверку доступа на уровне обработчика;
* has\_object\_permission() – проверяет доступ к объекту.

Чтобы разрешить доступ, метод должен возвращать True, а чтобы запретить – False.

**Создайте новый файл permissions.py в папке courses/api/** и добавьте в него такой код:

from rest\_framework.permissions import BasePermission

class IsEnrolled(BasePermission):

def has\_object\_permission(self, request, view, obj):

return obj.students.filter(id=request.user.id).exists()

Мы создали собственную реализацию разрешения. Клас IsEnrolled является

наслідником BasePermission и переопределяет метод has\_object\_permission(). Мы проверяем, является ли текущий пользователь слушателем курса, через атрибут students объекта модели Course. Чуть позже мы применим это разрешение.

**Сериализация содержимого курсов**

Для передачи данных с помощью Django REST нам необходимо сериализовать

содержимое курсов. Модель Content содержит обобщенную связь с несколькими моделями. В предыдущей главе мы уже добавили метод render() для каждого типа содержимого. Давайте задействуем его для формирования ответа на запросы к API.

**Отредактируйте файл api /serializers.py приложения courses и добавьте та-**

**кой код:**

from ..models import Content

class ItemRelatedField(serializers.RelatedField):

def to\_representation(self, value):

return value.render()

class ContentSerializer(serializers.ModelSerializer):

item = ItemRelatedField(read\_only=True)

class Meta:

model = Content

ᒀelds = ['order', 'item']

В этом фрагменте мы создали собственное поле, являющееся наследником

класса RelatedField фреймворка Django REST, и переопределили метод to\_representation(). Также мы создали сериализатор ContentSerializer для модели Content, в котором определили поле item типа ItemRelatedField.

Необходимо создать сериализатор для модели Module, который будет формировать данные входящего в модуль содержимого. Затем следует расширить сериализатор модели Course.

**Отредактируйте файл api /serializers.py** и добавьте в него такой код:

387

class ModuleWithContentsSerializer(serializers.ModelSerializer):

contents = ContentSerializer(many=True)

class Meta:

model = Module

ꌙelds = ['order', 'title', 'description', 'contents']

class CourseWithContentsSerializer(serializers.ModelSerializer):

modules = ModuleWithContentsSerializer(many=True)

class Meta:

model = Course

ꌙelds = ['id', 'subject', 'title', 'slug',

'overview', 'created', 'owner', 'modules']

Давайте реализуем обработчик, работающий аналогично методу retrieve() .

Он будет возвращать данные курса, его модулей и содержимого.

**Откройте файл api/views.py и измените класс CourseViewSet** таким образом:

from .permissions import IsEnrolled

from .serializers import CourseWithContentsSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):

# ...

@detail\_route(methods=['get'],

serializer\_class=CourseWithContentsSerializer,

authentication\_classes=[BasicAuthentication],

permission\_classes=[IsAuthenticated, IsEnrolled])

def contents(self, request, \*args, \*\*kwargs):

return self.retrieve(request, \*args, \*\*kwargs)

В этом методе мы выполняем следующие действия:

* оборачиваем метод в декоратор detail\_route, т. к. он работает с одним объектом;
* указываем, что метод обрабатывает только GET-запросы;
* обращаемся к сериализатору CourseWithContentsSerializer для формирования содержимого курса для ответа;
* используем разрешения IsAuthenticated и IsEnrolled. Так мы ограничим

доступ к курсам, и их содержимое смогут просматривать только записавшиеся студенты;

* вызываем существующий метод retrieve() , чтобы вернуть объект модели

Course.

Откройте в браузере страницу http: //127.0.0.1:8000/api/courses/1/contents/.

Если вы авторизованы как пользователь, зачисленный на курс, с идентификатором, равным 1, то увидите JSON-представление модулей и содержимого курса:

{

"order": 0,

"title": "Introduction to Django",

"description": "Brief introduction to the Django Web Framework.",

"contents": [

388

{

"order": 0,

"item": "<p>Meet Django. Django is a high-level

Python Web framework

...</p>"

},

{

"order": 1,

"item": "\n<iframe width=\"480\" height=\"360\"

src=\"http://www.youtube.com/embed/bgV39DlmZ2U?

wmode=opaque\"

frameborder=\"0\" allowfullscreen></iframe>\n"

}

]

}

Вы реализовали простой интерфейс к платформе онлайн-обучения, который

позволит другим системам получать данные о курсах. Фреймворк Django REST

содержит методы для создания и редактирования объектов с помощью набора

обработчиков ModelViewSet. Мы затронули основные и наиболее значимые на

практике стороны фреймворка Django REST. Ознакомиться с подробным описанием его возможностей и полной документацией вы можете на сайте https://www.django-rest-framework.org/.

**Резюме**

В этой главе мы создали RESTful API для взаимодействия приложения с други-

ми системами.

В следующей главе вы узнаете, как развернуть проект в боевом окружении

с помощью uWSGI и NGINX, а также реализуете собственный промежуточный

слой и создадите команду управления.