Geekbrains

**Разработка платформы для онлайн обучения на фреймворке**

**Django**

## IT-специалист:

## Программист Python Цифровые профессии

Крюкова Дарья Романовна

г. Санкт-Петербург

2024

Введение

Обоснование актуальности темы………………………………………………….……………..4

Цели и задачи исследования………………………………………………………..…………….4

Обзор существующих платформ для электронного обучения……………...…….……………5

Теоретическая часть

Определение понятий онлайн обучения и платформы для него………………….……………6

Анализ основных принципов и методов онлайн обучения……...…………….……………….6

Обзор современных технологий и инструментов для создания платформ для онлайн обучения……………………………………………………………………………………………………7

Разработка и реализация платформы для электронного обучения

Настройка проекта электронного обучения……………………………………………………10

Раздача медиафайлов………………………………………………….…………………………10

Разработка моделей курса……………………………………………….………………………11

Использование фикстур с целью предоставления моделям первоначальных данных…...…14

Создание моделей полиморфного содержимого………………………………………...…….16

Добавление представлений аутентификации…………………………………………………..22

Создание системы управления контентом

Создание CMS……………………………………………………………………………………26

Управление модулями курса и их содержимым……………………………………...…….….43

Переупорядочивание модулей и их содержимого……………………………….......……..….47

Прорисовка и кеширование контента

Отображение курсов……………………………………………………………………………..52

Добавление регистрации студентов………………………………………………………….…56

Доступ к содержимому курсов………………………………………………………………….61

Использование кеш-фреймворка…………………………………………….………………….65

Разработка API

Разработка RESTful API…………………………………………………………………………67

Разработка чат-сервера

Разработка чат-приложения с Django Channels…………………………………………….….76

Установка приложения-обертки Channels……………………………………...………………77

Пишем Потребителя……………………………………………………………………….…….78

Разбираемся в Маршрутизации для Веб-Сокетов в Django Channels…………………….…..79

Создание WebSocket Клиента в Django с JavaScript…………………………………………...80

Интеграция Канального Слоя для Мультиплексирования Сообщений……………...………84

Видоизменение потребителя с для обеспечения полной асинхронности……...…………….89

Интеграция приложения для ведения чата………………………………….…….……………91

Дальнейшее развитие проекта…………………………………………………………………….…….92

Заключение……………………………………………………………………………………………….93

Использованная литература ………………………………………………………………………….....94

Приложения………………………………………………………………………………………………96

**Введение**

**Обоснование актуальности темы**

Современный мир переживает быстрое развитие информационных технологий, которые проникают во все сферы жизни человека. Одним из ключевых направлений, где информационные технологии имеют огромный потенциал, является образование. В условиях быстрого темпа жизни и необходимости постоянного обучения, электронное обучение становится все более популярным и востребованным.

Разработка платформ для электронного обучения становится необходимой задачей для образовательных учреждений, компаний, тренинговых центров и других организаций, желающих предложить эффективные и удобные способы обучения своим сотрудникам или студентам. В современном мире, где удаленная работа и обучение становятся все более распространенными, платформы для электронного обучения играют ключевую роль в обеспечении доступности и качества образования.

Цель данной дипломной работы заключается в разработке современной и функциональной платформы для электронного обучения, которая будет соответствовать современным требованиям пользователей и обладать широким спектром возможностей. Таким образом, актуальность темы данной работы заключается в необходимости создания инновационных решений в области электронного обучения, которые помогут повысить эффективность образовательного процесса и улучшить доступность обучения для всех категорий пользователей.

Данное исследование представляет собой попытку создания современной платформы для электронного обучения, которая будет отвечать современным требованиям пользователей и образовательных учреждений. В дальнейшем разработка такой платформы может способствовать улучшению качества образования, повышению доступности обучения и развитию инновационных подходов в сфере образования.

Таким образом, данная дипломная работа имеет высокую актуальность в контексте современных вызовов и потребностей в области образования и электронного обучения.

**Цели и задачи исследования**

Цель работы: Разработать современную и функциональную платформу для электронного обучения, которая будет отвечать современным требованиям пользователей и образовательных учреждений.

Задачи исследования:

1. Изучить существующие платформы для электронного обучения и выявить их основные преимущества и недостатки.
2. Определить требования пользователей к платформе для электронного обучения.
3. Разработать дизайн и функционал платформы, учитывая современные тенденции в области образования и информационных технологий.
4. Создать прототип платформы для электронного обучения и провести тестирование его функционала.
5. Оценить эффективность разработанной платформы на основе обратной связи от пользователей и экспертов.
6. Предложить рекомендации по дальнейшему развитию и совершенствованию платформы для электронного обучения.

Выполнение поставленных задач позволит создать инновационное решение в области электронного обучения, которое будет способствовать повышению качества образования, улучшению доступности обучения и развитию современных подходов к обучению и развитию.

**Обзор существующих платформ для электронного обучения**

В современном мире существует множество платформ для электронного обучения, предлагающих различные возможности и функционал для учащихся и преподавателей. Ниже приведен краткий обзор нескольких популярных платформ:

1. Skillbox - платформа, специализирующаяся на курсах по дизайну, маркетингу, программированию и другим IT-направлениям.

2. Coursera - международная платформа, предлагающая курсы от ведущих университетов и компаний по различным тематикам.

3. GeekBrains - онлайн-школа, специализирующаяся на обучении программированию, дизайну и digital маркетингу.

4. OTUS - платформа для образования и развития профессиональных навыков, предлагающая курсы по IT, бизнесу, маркетингу и другим областям.

5. Stepik - платформа с бесплатными курсами по различным предметам, включая математику, программирование, физику и другие.

Каждая из этих платформ имеет свои особенности и преимущества, которые делают их популярными среди пользователей. При разработке новой платформы для онлайн обучения на фреймворке Django следует учитывать лучшие практики и функциональность уже существующих решений.

**Теоретическая часть**

**Определение понятий онлайн обучения и платформы для него**

Онлайн обучение - это форма обучения, которая осуществляется через интернет с использованием специальных платформ и ресурсов. Учащиеся могут изучать различные предметы, курсы и программы, а также проходить тесты и получать обратную связь от преподавателей или автоматизированных систем.

Платформа для онлайн обучения - это веб-сайт или приложение, предоставляющее учащимся доступ к образовательным материалам, курсам, тестам и другим ресурсам для обучения. Платформа может включать в себя функционал для регистрации студентов, создания и управления курсами, оценки успеваемости, коммуникации между участниками образовательного процесса и другие возможности, способствующие эффективному обучению.

Развитие онлайн обучения и платформ для него открывает новые возможности для людей, желающих получить знания и навыки в удобной форме, не выходя из дома. Платформы для онлайн обучения становятся все более популярными благодаря своей доступности, гибкости и широкому выбору образовательных материалов.

**Анализ основных принципов и методов онлайн обучения**

С развитием технологий и доступом к интернету онлайн обучение становится все более популярным и востребованным. Для удобства студентов и преподавателей создание платформы для онлайн обучения на фреймворке Django может быть эффективным решением. В данной работе будет проведен анализ основных принципов и методов онлайн обучения, а также описан процесс разработки платформы на основе фреймворка Django.

Анализ основных принципов онлайн обучения

1. Гибкость и доступность - студенты могут получать знания в удобное для них время и место.
2. Интерактивность - возможность взаимодействия с материалами, тестирование знаний и обратная связь.
3. Персонализация обучения - адаптация курсов под индивидуальные потребности и уровень знаний студентов.
4. Многообразие образовательных материалов - видеоуроки, тексты, тесты, задания.
5. Оценка успеваемости - возможность отслеживать прогресс студентов и оценивать их результаты.

Методы онлайн обучения:

1. Лекции и видеоуроки - предоставление информации в удобной для восприятия форме.
2. Тестирование и задания - проверка знаний студентов и оценка результатов.
3. Форумы и чаты - обсуждение материалов, вопросы студентов и обратная связь от преподавателей.
4. Интерактивные задания - задания, требующие активного участия студентов.

Разработка платформы на фреймворке Django подразумевает следующие этапы:

1. Создание базы данных для хранения информации о студентах, курсах, материалах.
2. Разработка пользовательского интерфейса для регистрации студентов, выбора курсов и прохождения обучения.
3. Создание системы управления контентом
4. Прорисовка и кеширование контента
5. Разработка API
6. Разработка чат-сервера

Разработка платформы для онлайн обучения на фреймворке Django позволяет учащимся получать знания эффективно и удобно. Анализ основных принципов и методов онлайн обучения помогает определить ключевые аспекты, необходимые для успешной реализации такой платформы. Важно учитывать потребности студентов и обеспечить доступ к разнообразным образовательным материалам для эффективного обучения.

**Обзор современных технологий и инструментов для создания платформ для онлайн обучения**

С развитием технологий и возрастанием интереса к онлайн обучению, создание платформ для обучения становится все более актуальным. В данной работе будет проведен обзор современных технологий и инструментов, которые могут быть использованы при разработке платформы для онлайн обучения на фреймворке Django. Анализ этих инструментов поможет определить оптимальный подход к созданию удобной и эффективной образовательной платформы.

Современные технологии и инструменты для создания платформ для онлайн обучения которые нам понадобятся:

1. Фреймворк **Django**: Django является мощным и гибким фреймворком для создания веб-приложений на языке Python. Он предоставляет широкий набор инструментов для разработки веб-приложений, включая работу с базами данных, аутентификацию пользователей, административные панели и многое другое.
2. Библиотека **Pillow**: Pillow используется для обработки изображений в приложении. Это позволяет загружать, изменять размеры и обрабатывать изображения для курсов и материалов на платформе.
3. Библиотека **sqlparse**: Sqlparse помогает форматировать SQL запросы для удобства чтения и отладки. Это полезный инструмент при работе с базой данных в приложении.
4. Библиотека **django-braces**: Django-braces предоставляет дополнительные классы Mixin для использования в представлениях Django. Это упрощает написание кода и повышает его читаемость.
5. Библиотека **django-embed-video**: Django-embed-video позволяет легко встраивать видео из различных источников на страницы платформы. Это может быть полезно для добавления видеоуроков или презентаций.
6. Библиотека **pymemcache**: Pymemcache используется для работы с кэшем данных в приложении. Кэширование может значительно ускорить доступ к данным и улучшить производительность.
7. Библиотека **django-debug-toolbar**: Django-debug-toolbar предоставляет инструменты для отладки приложения прямо в браузере. Это помогает быстро выявлять и исправлять ошибки.
8. Библиотека **redis** и **django-redisboard**: Redis используется как хранилище данных и кэш в приложении. Django-redisboard предоставляет административную панель для мониторинга и управления Redis.
9. Библиотека **djangorestframework**: Django REST framework используется для создания API на основе RESTful принципов. Это упрощает взаимодействие с платформой через стандартизированные HTTP запросы.
10. Библиотека **requests**: Requests используется для отправки HTTP запросов из приложения. Это может быть полезно при интеграции с другими сервисами или API.
11. Библиотеки **channels** и **channels-redis**: Channels позволяют создавать асинхронные приложения в Django, что полезно для реального времени обновлений на платформе. Channels-redis используется для работы с Redis в асинхронном режиме.
12. Библиотека **psycopg2**: Psycopg2 используется для работы с PostgreSQL базой данных в Django. Это расширение обеспечивает эффективное взаимодействие с PostgreSQL.
13. Серверы **uwsgi** и **daphne**: Uwsgi и Daphne используются как серверы приложений для запуска Django приложения. Они обеспечивают высокую производительность и масштабируемость.

Использование указанных современных технологий и инструментов при разработке платформы для онлайн обучения на фреймворке Django позволит создать функциональное, удобное и эффективное образовательное приложение. Оптимальный выбор библиотек и правильное применение инструментов помогут обеспечить высокую производительность, безопасность и удобство использования платформы как для студентов, так и для преподавателей.

**Разработка и реализация платформы для онлайн обучения**

**Настройка проекта электронного обучения**

Для начала активируем виртуальное окружение. В PyCharm она активируется самостоятельно при создании нового проекта.

Теперь установим Django в виртуальной среде **pip install Django.**

Также установим библиотеку Pillow, которая понадобится для управления загрузкой изображений pip install Pillow.

Следующим шагом будет создание нового проекта с помощью команды startproject. После этого войдем в созданный каталог education и создадим новое приложение courses:

django-admin startproject education

cd education

django-admin startapp courses

Далее добавим приложение courses в настроечный параметр INSTALLED\_APPS в файле settings.py проекта education. Новая строка будет выглядеть примерно так:

INSTALLED\_APPS = [

'courses.apps.CoursesConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Теперь приложение courses активно, и мы готовы подготовить проект к работе с медиафайлами и созданию моделей курсов и контента курсов.

**Раздача медиафайлов**

Перед тем, чем мы начнем создавать модели курсов и содержимого курсов, нужно подготовить наш проект к раздаче медиафайлов. Это важно, поскольку преподаватели курсов должны иметь возможность загружать медиафайлы в содержимое курсов через систему управления контентом (CMS), которую мы планируем создать. Для этого мы проведем некоторые конфигурационные изменения, чтобы проект мог обслуживать медиафайлы.

Начнем с редактирования файла settings.py в нашем проекте. Добавим следующие строки:

MEDIA\_URL = 'media/'

MEDIA\_ROOT = BASE\_DIR / 'media'

Это позволит Django управлять загрузкой файлов и обслуживать медиафайлы. MEDIA\_URL представляет базовый URL для обслуживания загруженных пользователями медиафайлов, а MEDIA\_ROOT указывает локальный путь к местоположению этих файлов. Пути и URL'ы файлов создаются динамически путем добавления к ним пути проекта или URL'а медиафайла в качестве префикса для обеспечения переносимости.

После этого нам нужно внести изменения в основной файл urls.py в нашем проекте education, как показано ниже. Новые строки выделены жирным шрифтом:

from django.contrib import admin

from django.urls import path

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

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

]

if settings.DEBUG:

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

Здесь мы добавляем вспомогательную функцию static(), чтобы раздавать медиафайлы встроенным в Django сервером разработки во время разработки, что актуально, когда значение настроечного параметра DEBUG установлено равным True.

Таким образом, после проведенных изменений, наш проект готов к работе с медиафайлами. Теперь мы можем приступить к созданию моделей для курсов и их содержимого.

**Разработка моделей курса**

Каждый курс будет состоять из настраиваемого числа модулей, а каждый модуль будет содержать различные типы контента, такие как текст, файлы, изображения или видео.

Наши модели хранятся в файле models.py, который нужно отредактировать, чтобы добавить некоторый исходный код. Давайте разберем, что там происходит.

class Subject(models.Model):

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):

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):

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

Этот код представляет собой первоначальные модели Subject и Course. Модель Course имеет такие поля как "owner", "subject", "title", "slug", "overview" и "created". Простите за подробный разбор, но важно, чтобы все было понятно.

Поле "owner" ссылается на пользователя, создавшего курс. Поле "subject" является внешним ключом, указывающим на модель Subject, и представляет собой предмет, к которому относится курс. Поля "title" и "slug" отвечают за название и уникальный слаг курса соответственно. "Overview" представляет собой краткое описание курса, а "created" будет автоматически заполняться датой и временем создания, благодаря auto\_now\_add=True.

Каждый курс состоит из нескольких модулей, поэтому модель Module содержит поле ForeignKey, указывающее на модель Course.

Этот класс имеет поля, такие как "course", "title" и "description". Поле "course" является внешним ключом, указывающим на модель Course. related\_name='modules' будет использоваться для обратного доступа к модулям из модели Course. Опция on\_delete=models.CASCADE указывает на то, что при удалении связанного курса, все модули, относящиеся к этому курсу, также будут удалены.

"Title" представляет собой название модуля, а "description" - необязательное поле, где вы можете описать содержание модуля.

Пoсле этого определен метод \_\_str\_\_, который возвращает название модуля.

Затем нужно выполнить команду **python manage.py makemigrations** для создания первоначальной миграции этого приложения.

После создания миграции, необходимо выполнить команду **python manage.py migrate** для применения всех миграций к базе данных.

Мы через код добавим модели курса на сайт администрирования в Django. Для этого нам нужно открыть файл admin.py внутри каталога приложения courses и добавить туда следующий код:

@admin.register(Subject)

class SubjectAdmin(admin.ModelAdmin):

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

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

class ModuleInline(admin.StackedInline):

model = Module

@admin.register(Course)

class CourseAdmin(admin.ModelAdmin):

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

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

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

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

inlines = [ModuleInline]

Код выше регистрирует модели Subject и Course на сайте администрирования. Декоратор @admin.register() используется для их регистрации.

В резюме, этот код делает следующее:

1. Регистрирует модель Subject с пользовательским списком отображения и автозаполнением поля "slug".
2. Затем создает встраиваемую модель ModuleInline для модели Module.
3. После этого регистрирует модель Course с пользовательским списком отображения, фильтрами и возможностью поиска, также с автозаполнением поля "slug" и встраиваемой моделью ModuleInline для управления модулями курса.

**Использование фикстур с целью предоставления моделям первоначальных данных**

Что такое фикстуры и зачем они нужны:

Итак, иногда возникает ситуация, когда нужно заполнить базу данных заранее заданными данными. Это может быть удобно для автоматической настройки проекта без ручного добавления данных. В Django есть удобный способ загрузки и выгрузки данных из базы, который называется фикстуры. Эти данные могут быть в форматах JSON, XML или YAML.

Как создать фикстуру для модели Subject:

1. Сначала создай суперпользователя, используя команду:
2. python manage.py createsuperuser
3. Затем запусти сервер разработки:
4. python manage.py runserver
5. Перейди по URL-адресу <http://127.0.0.1:8000/admin/courses/subject/> в браузере и создай несколько предметов. **Рис.1**
6. После этого выполните команду из оболочки:
7. python manage.py dumpdata courses --indent=2

Команда dumpdata выгружает данные из базы данных в стандартный вывод, по умолчанию в формате JSON. Результат представляет собой данные о модели и ее полях, что позволяет Django загрузить их обратно в базу данных.

[

{

"model": "courses.subject",

"pk": 1,

"fields": {

"title": "Frontend Developer",

"slug": "frontend-developer"

}

},

{

"model": "courses.subject",

"pk": 2,

"fields": {

"title": "Full stack developer",

"slug": "full-stack-developer"

}

},

{

"model": "courses.subject",

"pk": 3,

"fields": {

"title": "Project Manager",

"slug": "project-manager"

}

},

{

"model": "courses.subject",

"pk": 4,

"fields": {

"title": "Interface designer",

"slug": "interface-designer"

}

},

{

"model": "courses.subject",

"pk": 5,

"fields": {

"title": "Internet Marketer",

"slug": "internet-marketer"

}

}

]

Сохранение дампа в виде фикстуры:

1. Первым делом мы создаем новый каталог fixtures/ внутри нашего приложения courses с помощью команды
2. mkdir courses/fixtures
3. Затем мы используем команду
4. python manage.py dumpdata courses --indent=2 --output=courses/fixtures/subjects.json
5. для сохранения дампа данных в файл-фикстуру. Здесь мы получаем файл subjects.json, содержащий данные для модели Subject.

Удаление и загрузка фикстур:

После сохранения фикстуры шагнем дальше. Мы запускаем сервер разработки и заходим на сайт администрирования, чтобы удалить созданные нами предметы, используя соответствующий интерфейс. **Рис.2**

После удаления всех предметов, мы загружаем фикстуру в базу данных с помощью команды python manage.py loaddata subjects.json. **Рис.3**

Это позволит нам восстановить ранее удаленные объекты Subject, используя файл-фикстуру. **Рис.4**

**Создание моделей полиморфного содержимого**

Мы хотим добавлять разные типы содержимого, такие как текст, изображения, файлы и видео, в модули курса. Давай создадим модель, которая будет представлять содержимое модулей, и определим обобщенное отношение, чтобы связывать любой объект с объектом Content.

Имплементация модели Content:

Для начала добавим следующие импорты в файл models.py вашего приложения courses:

from django.contrib.contenttypes.models import ContentType

from django.contrib.contenttypes.fields import GenericForeignKey

Затем добавим следующий код в конец файла models.py:

class Content(models.Model):

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')

Что делает эта модель:

1. В этой модели Content содержится поле ForeignKey, указывающее на модель Module, а также обобщенное отношение, которое позволяет связывать объекты из разных моделей, представляющих разные типы содержимого.
2. Для установления обобщенного отношения требуется три разных поля:

* content\_type: поле ForeignKey для модели ContentType
* object\_id: поле PositiveIntegerField для хранения первичного ключа связанного объекта
* item: поле GenericForeignKey для связанного объекта, объединяющее два предыдущих поля

Использование разных моделей для разных типов содержимого:

Мы будем использовать разные модели для каждого типа содержимого, где эти модели будут иметь несколько общих полей, но будут отличаться фактическими данными, которые они могут хранить, таким образом создавая единый интерфейс для разных типов содержимого.

**Создание моделей Content**

Для каждого типа содержимого у нас будет своя модель. Все модели Content будут иметь некоторые общие поля, а также дополнительные поля для конкретных прикладных данных.

Мы создадим абстрактную модель, которая будет предоставлять общие поля для всех моделей Content. Вот как это выглядит:

from django.db import models

from django.contrib.auth.models import User

class ItemBase(models.Model):

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):

content = models.TextField()

class File(ItemBase):

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

class Image(ItemBase):

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

class Video(ItemBase):

url = models.URLField()

В этом исходном коде мы определяем абстрактную модель под названием ItemBase. Устанавливаем abstract=True в классе Meta, чтобы указать, что это абстрактная модель. Мы также определяем поля owner, title, created и updated, которые будут использоваться для всех типов содержимого.

Поле owner позволяет нам хранить пользователя, создавшего контент. Поскольку это поле определено в абстрактной модели, каждая дочерняя модель потребует другое имя related\_name. Django позволяет использовать заполнители в атрибуте related\_name, такие как %(class)s, что позволяет генерировать соответствующие имена автоматически для каждой дочерней модели.

Так, у нас есть четыре разные модели Content: Text, File, Image, и Video, и все они наследуют от абстрактной модели ItemBase. Каждая из этих дочерних моделей содержит свои собственные поля, а также поля, определенные в ItemBase.

Вот как происходит обсуждение в коде:

content\_type = models.ForeignKey(

ContentType,

on\_delete=models.CASCADE,

limit\_choices\_to={'model\_\_in':( 'text', 'video', 'image', 'file' )})

В этом фрагменте кода мы добавляем аргумент limit\_choices\_to к полю content\_type, чтобы ограничить объекты ContentType, которые могут быть использованы в обобщенном отношении. Мы используем выражение поиска в model\_\_in, чтобы фильтровать запрос объектами ContentType с атрибутом model, содержащим 'text', 'video', 'image' или 'file'.

Теперь у нас есть модели, которые могут использоваться для добавления разнообразного контента в модули курса. Однако у нас всё ещё не хватает чего-то важного: модули и содержимое курса должны следовать определенному порядку. Нам нужно поле, которое позволит легко упорядочивать их.

**Создание конкретно-прикладных модельных полей**

В нашем проекте нам нужно поле, которое будет устанавливать порядок следования объектов. Делать это можно с помощью стандартного поля PositiveIntegerField, используя целые числа для обозначения порядка следования. Однако мы хотим создать конкретно-прикладное поле для порядка, которое будет наследоваться от PositiveIntegerField и добавлять дополнительное функциональное поведение.

В новое поле порядка будут встроены две соответствующие функиции:

1. Автоматическое назначение порядка следования, если явно не указан конкретный порядок: при сохранении нового объекта без указания порядка, поле будет автоматически назначать число, следующее за последним упорядоченным объектом. Например, при наличии объектов с порядковыми номерами 1 и 2, при сохранении третьего объекта ему автоматически будет присвоен порядок 3.
2. Возможность упорядочивать объекты по другим полям: модули курса будут упорядочиваться по курсу, к которому они принадлежат, а содержимое модуля по отношению к модулю, к которому они принадлежат.

Для этого мы создаем новый файл fields.py внутри каталога приложения courses и добавляем в него следующий код для нового поля OrderField:

from django.db import models

from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):

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

self.for\_fields = for\_fields

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

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

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

# текущее значение отсутствует

try:

qs = self.model.objects.all()

if self.for\_fields:

query = {field: getattr(model\_instance, field) for field in self.for\_fields}

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

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().pre\_save(model\_instance, add)

Это новое конкретно-прикладное поле OrderField, которое наследуется от стандартного поля PositiveIntegerField в Django. Поле OrderField принимает необязательный параметр for\_fields, который позволяет указывать поля, используемые для упорядочивания данных.

Это поле переопределяет метод pre\_save() поля PositiveIntegerField, который вызывается перед сохранением поля в базе данных. В этом методе происходит автоматическое определение значения порядка следования объектов, если оно не было указано явно.

Давай подробно разберем, что именно происходит в этом методе.

1. Проверка наличия значения и вычисление порядка:

* Мы проверяем, есть ли уже значение у этого поля в экземпляре модели, используя self.attname – это имя атрибута, присвоенное полю в модели.
* Если значение атрибута не равно None, это означает, что нам необходимо вычислить порядок, который ему следует присвоить. Это выполняется следующим образом:

1. Мы создаем набор запросов QuerySet для извлечения всех объектов данной модели. Обращаясь к self.model, мы извлекаем класс модели, к которой относится данное поле.
2. Если в атрибуте for\_fields поля есть какие-либо имена полей, мы фильтруем этот набор запросов QuerySet по текущим значениям модельных полей, указанных в for\_fields. Это позволяет нам вычислить порядок относительно указанных полей.
3. С помощью last\_item = qs.latest(self.attname) мы извлекаем из базы данных объект с наивысшим порядком. Если объект не найден, мы предполагаем, что это первый объект, и ему присваивается порядок 0.
4. Если объект найден, к его порядку добавляется 1.
5. С помощью setattr() рассчитанный порядок присваивается значению поля в экземпляре модели, и затем этот порядок возвращается.

2. Использование существующего значения, если оно есть:  
 Если у текущего поля экземпляра модели уже есть значение, мы используем это значение вместо вычисления нового порядка.

**Добавление упорядочивания в модули и объекты содержимого**

Для начала нам нужно отредактировать файл models.py в приложении courses, чтобы импортировать класс OrderField и добавить поле в модель Module:

from .fields import OrderField

class Module(models.Model):

# ...

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

Здесь мы задаем новый порядок для полей, указав, что порядок вычисляется относительно курса, используя for\_fields=['course']. Это означает, что порядок нового модуля будет назначаться путем прибавления 1 к последнему модулю того же объекта Course.

Теперь давайте отредактируем метод \_\_str\_\_() для модели Module, чтобы включить в него порядок:

class Module(models.Model):

# ...

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

return f'{self.order}. {self.title}'

Теперь, чтобы упорядочить содержимое модуля, мы добавим поле OrderField в модель Content:

class Content(models.Model):

# ...

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

Здесь мы указываем, что порядок вычисляется относительно поля module.

Наконец, давайте добавим порядок, который будет применяться по умолчанию для обеих моделей, добавив следующий Meta-класс в модели Module и Content:

class Module(models.Model):

# ...

class Meta:

ordering = ['order']

class Content(models.Model):

# ...

class Meta:

ordering = ['order']

Теперь наши модели Module и Content выглядят следующим образом:

class Module(models.Model):

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 f'{self.order}. {self.title}'

class Content(models.Model):

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']

Теперь, после всех этих шагов, у нас есть модели Module и Content, которые упорядочены соответственно. Это позволит легко управлять порядком модулей и содержимого, делая их более структурированными и организованными.

С помощью ORM создадим курс и добавим модули. **Рис.5 ,Рис.6**

**Добавление представлений аутентификации**

После того, как мы создали мощную полиморфную модель данных для управления курсами, мы теперь строим систему управления контентом (CMS), чтобы управлять курсами и их содержимым.

Этап номер один - добавим систему аутентификации в CMS. Мы собираемся использовать встроенный в Django фреймворк аутентификации для проверки подлинности пользователей на нашей платформе онлайн-обучения. И, заметь, как удивительно, что и преподаватели, и студенты могут быть пользователями модели User, предоставляемой Django. И это означает, что они могут входить на сайт, используя представления аутентификации из django.contrib.auth.

Внесем изменения в основной файл urls.py проекта educa, чтобы добавить представления входа (login) и выхода (logout) из встроенного фреймворка аутентификации Django:

from django.contrib import admin

from django.urls import path

from django.conf import settings

from django.conf.urls.static import static

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

urlpatterns = [

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

path('accounts/logout/', auth\_views.LogoutView.as\_view(), name='logout'),

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

]

if settings.DEBUG:

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

**Создание шаблонов аутентификации**

Подготовим базовый шаблон проекта.

Создадим следующую структуру:

templates/

base.html

registration/

login.html

logged\_out.html

Откроем файл base.html в каталоге шаблонов и добавим следующее содержимое:

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8"/>

<title>{% block title %}Education{% endblock %}</title>

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

</head>

<body>

<div id="header">

<a href="/" class="logo">Education</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>

document.addEventListener('DOMContentLoaded', (event) => {

// DOM-модель загружена

{% block domready %}

{% endblock %}

})

</script>

</body>

</html>

Этот базовый шаблон будет расширяться другими шаблонами, и в нем уже определены несколько блоков:

* title: блок для других шаблонов, где они могут добавить свой заголовок для каждой страницы.
* content: основной блок содержимого. Все шаблоны, расширяющие этот базовый шаблон, должны добавлять содержимое в этот блок.
* domready: находится внутри прослушивателя событий JavaScript, который отслеживает событие DOMContentLoaded, позволяя выполнять JavaScript-код после загрузки DOM.

Внутри шаблона registration/login.html ты расширяешь базовый шаблон и добавляешь содержимое для страницы входа. Вот что ты добавляешь:

{% 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 %}

Здесь мы создаём простую форму входа пользователя, обрабатывая возможные ошибки, которые могут возникнуть при вводе данных пользователя.

В шаблоне registration/logged\_out.html, ты также расширяешь базовый шаблон и добавляешь содержимое для страницы "Выход выполнен":

{% 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 %}

Теперь, когда есть эти шаблоны, мы можем перейти к запуску сервера разработки. Запустить его с помощью команды **python manage.py runserver** и затем пройдём по адресам <http://127.0.0.1:8000/accounts/login/> и <http://127.0.0.1:8000/accounts/logout/> в своем браузере. Мы увидим страницы входа и выхода из учетной записи, которые создали. **Рис.7**

**Создание системы управления контентом**

**Создание CMS**

Мы готовимся создать систему управления контентом (CMS) после того, как создали различные модели данных. Теперь пред дверью - создание инструмента, который позволит преподавателям создавать курсы и редактировать их содержимое. Вот что мы хотим реализовать:

* Показывать список курсов, созданных преподавателями.
* Давать возможность создавать, редактировать и удалять курсы.
* Добавлять модули в курсы и иметь возможность изменять их порядок.
* Добавлять разнообразное содержимое в каждый модуль.
* Иметь возможность переупорядочивать модули и содержимое курса.

**Создание представлений на основе классов**

Мы собираемся создать представления для создания, редактирования и удаления курсов, используя представления на основе классов. Для начала давай посмотрим на файл views.py в приложении courses. Вот что мы добавляем:

from django.views.generic.list import ListView

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

from django.urls import reverse\_lazy

from .models import Course

class ManageCourseListView(ListView):

model = Course

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

def get\_queryset(self):

qs = super().get\_queryset()

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

Это представление ManageCourseListView. Оно использует наследование от встроенного в Django типового представления ListView. Мы переопределяем метод get\_queryset() представления, чтобы извлечь только те курсы, которые были созданы текущим пользователем.

Также, важно отметить, что нужно переопределить метод get\_queryset() в представлениях Create, Update и Delete для предотвращения пользователей от редактирования, обновления или удаления курсов, которые они не создали.

**Использование примесей для представлений на основе классов**

Примеси - это способ множественного наследования для добавления дополнительной функциональности классам. Они предоставляют возможность добавить особые возможности при объединении с другими примесями, формируя поведение класса в соответствии с нашими потребностями. Обычно примеси используются в двух основных ситуациях: в случаях, когда необходимо дать классу несколько дополнительных функций на выбор, или при необходимости использовать одну функциональность в нескольких классах.

Django предоставляет несколько встроенных примесей, которые расширяют функциональность представлений на основе классов.

В коде мы создадим общее поведение для нескольких представлений в примесных классах, которые затем будем использовать в представлениях для курсов.

from django.views.generic.list import ListView

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

from django.urls import reverse\_lazy

from .models import Course

class OwnerMixin:

def get\_queryset(self):

qs = super().get\_queryset()

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

class OwnerEditMixin:

def form\_valid(self, form):

form.instance.owner = self.request.user

return super().form\_valid(form)

class OwnerCourseMixin(OwnerMixin):

model = Course

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

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

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):

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

class ManageCourseListView(OwnerCourseMixin, ListView):

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

class CourseCreateView(OwnerCourseEditMixin, CreateView):

pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):

pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):

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

В представленном коде примеси OwnerMixin и OwnerEditMixin, которые будут использоваться в сочетании с встроенными представлениями Django, такими как ListView, CreateView, UpdateView и DeleteView. OwnerMixin реализует метод get\_queryset(), который используется представлениями для получения базового набора запросов QuerySet. Этот примесный класс будет переопределять этот метод, чтобы отфильтровать объекты по атрибуту владельца (owner), извлекая объекты, принадлежащие текущему пользователю (request.user).

Класс OwnerEditMixin. Он внедряет магию в метод form\_valid(). Для тех, кто не в курсе, этот метод используется в представлениях, которые используют Django ModelFormMixin.

Когда форма, отправленная на обработку, проходит валидацию, метод form\_valid() вступает в действие. По умолчанию он сохраняет экземпляр (в случае модельных форм) и перенаправляет пользователя на адрес Success\_url. Но, осторожно, здесь есть хитрость: мы можем переписать этот метод, чтобы автоматически устанавливать текущего пользователя в качестве владельца сохраняемого объекта. Таким образом, при сохранении объекта автоматически устанавливается его владелец.

Теперь погрузимся в OwnerMixin. Этот класс можно использовать в представлениях, которые взаимодействуют с любой моделью, содержащей атрибут owner.

OwnerCourseMixin. Он унаследован от OwnerMixin и предоставляет атрибуты для дочерних представлений. Он определяет модель, используемую для наборов запросов, поля модели для компоновки модельной формы, и Success\_url, который используется для перенаправления пользователя после успешной передачи формы.

OwnerCourseEditMixin! Этот класс определяет атрибут template\_name, который используется для представлений CreateView и UpdateView.

Давайте перейдем к созданию представлений. У нас есть четыре новых представления:

* ManageCourseListView: выводит список созданных пользователем курсов. Наследуется от OwnerCourseMixin и ListView. Он имеет свой собственный шаблон для отображения списка курсов.
* CourseCreateView: использует модельную форму для создания нового объекта Course. Использует поля, определенные в OwnerCourseMixin, и имеет свой шаблон, определенный в OwnerCourseEditMixin.
* CourseUpdateView: предоставляет возможность редактировать существующий объект Course. Использует поля из OwnerCourseMixin и имеет свой шаблон из OwnerCourseEditMixin.
* CourseDeleteView: наследует от OwnerCourseMixin и типового DeleteView. Определяет свой собственный шаблон для подтверждения удаления курса.

Займемся аутентификационными группами и разрешениями Django, чтобы регулировать доступ к этим представлениям.

**Работа с группами и разрешениями**

Сейчас все пользователи имеют доступ к представлениям для управления курсами, но нам нужно ограничить этот доступ так, чтобы только преподаватели имели право создавать и управлять ими. И здесь на помощь приходят модели аутентификации и разрешений в Django.

Давайте начнем с создания группы для пользователей-преподавателей и назначения им соответствующих разрешений. После этого мы запустим сервер разработки с помощью следующей команды: **python manage.py runserver**

Теперь откроем браузер и перейдем по адресу <http://127.0.0.1:8000/admin/auth/group/add/> для создания новой группы. Мы дадим ей имя "Instructors" и выберем все разрешения для приложения courses, за исключением разрешений модели Subject, как показано на **Рис. 8**.

Для каждой модели существует четыре различных типа разрешений: можно просматривать, добавлять, изменять и удалять. После выбора разрешений для этой группы мы нажимаем кнопку "SAVE" (Сохранить).

Django автоматически создает разрешения для моделей, но помимо этого мы можем создавать конкретные прикладные разрешения. Далее мы также научимся создавать такие конкретные разрешения.

Теперь давайте зайдем на страницу <http://127.0.0.1:8000/admin/auth/user/add/> и создадим нового пользователя. Затем мы отредактируем этого пользователя, добавив его в группу Instructors, как показано **Рис.9**.

Пользователи наследуют разрешения от групп, к которым они принадлежат, но помимо этого мы также можем добавлять индивидуальные разрешения каждому пользователю через административный сайт Django. Также стоит отметить, что пользователи, у которых значение параметра is\_superuser установлено в True, автоматически получают все разрешения.

**Ограничение доступа к представлениям, основанным на классах**

Мы стремимся ограничить доступ к представлениям, чтобы только пользователи с соответствующими разрешениями могли добавлять, изменять или удалять объекты курса. Это важно для обеспечения безопасности и предоставления точного управления.

Итак, для этой цели вы используем два крутых примесных класса, предоставленных фреймворком django.contrib.auth:

* LoginRequiredMixin: воспроизводит функциональность декоратора login\_required;
* PermissionRequiredMixin: предоставляет доступ к представлению пользователям с конкретным разрешением. Напоминаю, что суперпользователи автоматически получают все разрешения.

Давайте добавим этот функционал в наш код. Сначала проводим импорт этих примесных классов в файле views.py нашего приложения courses:

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

И затем повесим эти примесные классы на наш основной примесный класс OwnerCourseMixin:

class OwnerCourseMixin(OwnerMixin, LoginRequiredMixin, PermissionRequiredMixin):

model = Course

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

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

Теперь, в наших представлениях курса, добавим атрибут permission\_required, чтобы установить необходимое разрешение для каждой операции:

class ManageCourseListView(OwnerCourseMixin, ListView):

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

permission\_required = 'courses.view\_course'

class CourseCreateView(OwnerCourseEditMixin, CreateView):

permission\_required = 'courses.add\_course'

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):

permission\_required = 'courses.change\_course'

class CourseDeleteView(OwnerCourseMixin, DeleteView):

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

permission\_required = 'courses.delete\_course'

Примесный класс PermissionRequiredMixin убедится, что обращающийся к представлению пользователь имеет разрешение, указанное в атрибуте permission\_required. Это гарантирует, что созданные представления становятся доступны только пользователям с соответствующими разрешениями.

Теперь нам нужно создать URL-адреса для этих представлений. Создадим файл urls.py внутри каталога нашего приложения courses и добавим следующий код:

from django.urls import path

from . import views

urlpatterns = [

path('mine/', views.ManageCourseListView.as\_view(), name='manage\_course\_list'),

path('create/', views.CourseCreateView.as\_view(), name='course\_create'),

path('<pk>/edit/', views.CourseUpdateView.as\_view(), name='course\_edit'),

path('<pk>/delete/', views.CourseDeleteView.as\_view(), name='course\_delete'),

]

Это шаблоны URL-адресов для вывода списка, создания, редактирования и удаления курсов. Параметр pk относится к полю первичного ключа. Каждая модель Django имеет поле, которое служит первичным ключом. Обычно это поле с именем id.

Наконец, отредактируем главный файл urls.py проекта education, добавив шаблоны URL-адресов приложения courses:

from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.conf.urls.static import static

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

urlpatterns = [

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

path('accounts/logout/', auth\_views.LogoutView.as\_view(), name='logout'),

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

path('course/', include('courses.urls')), # Новый исходный код

]

if settings.DEBUG:

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

Для этих важных представлений необходимо создать соответствующие шаблоны. Для начала мы создаем несколько каталогов и файлов шаблонов внутри каталога templates нашего приложения courses, чтобы все было аккуратно и организованно.

Вот что нам нужно сделать:  
1. Создать следующие каталоги и файлы:

courses/

manage/

course/

list.html

form.html

delete.html

2. Теперь отредактируем шаблон courses/manage/course/list.html, вставив в него следующий код:

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

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

</p>

</div>

{% empty %}

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

{% endfor %}

<p>

<a href="{% url " course\_create" %}" class="button">Create new course</a>

</p>

</div>

{% endblock %}

Этот шаблон представляет ManageCourseListView. Он перечисляет курсы, созданные текущим пользователем, вставляя ссылки на редактирование или удаление каждого курса, а также ссылку на создание новых курсов.

3. Теперь создадим шаблон для формы создания и обновления представлений курса. Откройте form.html и напишите следующий 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 method="post">

{{ form.as\_p }}

{% csrf\_token %}

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

</form>

</div>

{% endblock %}

Этот шаблон используется для представлений CourseCreateView и CourseUpdateView. В нем выполняется проверка наличия переменной object в контексте. Если object существует, значит, обновляется существующий курс, и используется в заголовке страницы; в противном случае создается новый объект Course.

Как только все это будет готово, перезапусти сервер разработки и пройди по URL-адресу <http://127.0.0.1:8000/accounts/login/?next=/course/mine/> в браузере. После входа под пользователем, принадлежащим к группе Instructors, тебе будет показана страница курсов преподавателя, где отобразятся все твои созданные курсы. **Рис.10**

Итак, отправляемся по ссылке <http://127.0.0.1:8000/course/mine/>в своем браузере, а затем нажмём на кнопку CREATE NEW COURSE (Создать новый курс). Тебя ждет страница, которую явно показывают на **Рис.11**.

Там заполни форму, нажми SAVE COURSE (Сохранить курс), и вуаля! Твой курс сохранен, а ты перенаправлен на страницу со списком курсов. Пейзаж должен выглядеть красочно, как показано на **Рис.12**.

После этого, кликни по ссылке Edit (Редактировать) для только что созданного курса. Опять-таки, тебе отобразится форма, но на этот раз ты будешь редактировать существующий объект Course, а не создавать новый.

И вот почти заключительный этап! Необходимо отредактировать шаблон courses/manage/course/delete.html, добавив в него следующий код:

{% 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" value="Confirm">

</form>

</div>

{% endblock %}

Этот шаблон представляет CourseDeleteView. Это представление наследует встроенное в Django представление DeleteView и требует от пользователя подтверждения на удаление объекта.

После того как ты откроешь список курсов в браузере и кликнешь по ссылке Delete (Удалить) курса, откроется страница подтверждения, похожая на то, что изображено на **Рыс.13**.

Кликни по кнопке CONFIRM (Подтвердить), и курс будет удален, а затем ты вернешься на страницу со списком курсов.

На этом преподаватели обзавелись полной властью: они теперь могут создавать, редактировать и удалять курсы.

**Использование наборов форм для модулей курса**

Вначале, для работы с несколькими формами на одной странице в Django используются наборы форм, или formsets. Они предназначены для управления несколькими экземплярами форм определенного класса Form или ModelForm.

Основные особенности наборов форм:

* Одновременная валидация: Наборы форм содержат метод is\_valid() для одновременной валидации всех форм, позволяя обрабатывать их все сразу.
* Ограничение количества: Набор форм также ограничивает количество отображаемых и обрабатываемых форм, что делает их очень удобными для динамического управления количеством форм.

Оформим это в коде. Для начала создадим файл forms.py внутри каталога нашего приложения courses и добавим в него следующий 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

)

В чем здесь основные моменты:

1. Мы использовали функцию inlineformset\_factory, предоставляемую Django, для создания набора форм ModuleFormSet.
2. inlineformset\_factory позволяет динамически создавать наборы форм для связанных объектов на основе модели.
3. Мы указали fields, которые будут включены в каждую форму набора форм, а также настроили параметры extra и can\_delete.

* fields: определяет, какие поля будут включены в каждую форму набора форм.
* extra: устанавливает количество дополнительных пустых форм для отображения в наборе форм.
* can\_delete: если установлено в True, Django вставит булево поле для каждой формы, что позволит помечать объекты, которые могут быть удалены.

Давай рассмотрим представление CourseModuleUpdateView, которое отвечает за обработку набора форм, предназначенного для добавления, обновления и удаления модулей для конкретного курса.

Первым делом обратим внимание на следующий код в файле views.py внутри нашего приложения courses:

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

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

class CourseModuleUpdateView(TemplateResponseMixin, View):

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

course = None

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

return ModuleFormSet(instance=self.course,

data=data)

def dispatch(self, request, pk):

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

id=pk,

owner=request.user)

return super().dispatch(request, pk)

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

formset = self.get\_formset()

return self.render\_to\_response({

'course': self.course,

'formset': formset})

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

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})

Это представление наследует от следующих примесей и представлений:

* TemplateResponseMixin: эта примесь отвечает за прорисовку шаблонов и возврат HTTP-ответа. Нам нужно указать атрибут template\_name, который указывает, какой шаблон использовать, а также предоставить метод render\_to\_response(), чтобы передать контекст и прорисовать шаблон.
* View: базовое представление на основе класса, предоставляемое веб-фреймворком Django. В данном представлении реализованы методы get() и post() для обработки запросов методами GET и POST соответственно.

В представлении CourseModuleUpdateView были реализованы несколько методов для обработки запросов и управления формами.

Давай начнем с метода get\_formset(), который создает объект ModuleFormSet для определенного объекта Course. Этот метод позволяет избежать повторения кода при компоновке набора форм, обеспечивая гибкую настройку данных для формы.

Затем идет метод dispatch(). В данном случае, этот метод принимает HTTP-запрос и параметры, затем делегирует его соответствующему методу (get() или post()), в зависимости от используемого HTTP-метода. Внутри dispatch() мы используем функцию get\_object\_or\_404() для извлечения объекта Course на основе параметра id, принадлежащего текущему пользователю. Этот код вставляется в dispatch(), потому что требуется извлекать курс как для запросов GET, так и для запросов POST, и сохранять его в атрибуте course, чтобы сделать его доступным для других методов.

Далее, метод get() выполняется для запросов методом GET. Здесь создается пустой набор форм ModuleFormSet и прорисовывается по шаблону вместе с текущим объектом Course, используя метод render\_to\_response() предоставленный примесным классом TemplateResponseMixin.

Наконец, метод post() обрабатывает запросы методом POST. В этом методе создается экземпляр ModuleFormSet, используя отправленные на обработку данные. Затем выполняется метод is\_valid() для валидации всех форм в наборе. Если набор форм валиден, то он сохраняется, вызывая метод save(), после чего пользователи перенаправляются на URL-адрес manage\_course\_list. В случае, если набор форм невалиден, шаблон прорисовывается для отображения ошибок.

Эти методы позволяют удобно и эффективно работать с формами и управлять модулями курса в наших представлениях.

Теперь нам нужно отредактировать файл urls.py и создать шаблоны.

Давай начнем с файла urls.py courses. Здесь мы добавляем новый шаблон URL-адреса, чтобы управлять обновлением модулей курса. Тут важно, чтобы сервер знал, как обрабатывать запросы, связанные с обновлением модулей курса.

from django.urls import path

from . import views

urlpatterns = [

# ... другие URL-адреса ...

path('<pk>/module/', views.CourseModuleUpdateView.as\_view(), name='course\_module\_update'),

]

Теперь давайте перейдем к созданию нового каталога и шаблона. Мы создадим каталог внутри templates/ приложения courses создайте следующие ниже каталоги и файлы:

courses/

manage/

module/

Затем в этом каталоге создадим файл formset.html, в который поместим следующий HTML-код:

{% extends "base.html" %}

{% block title %}

Edit "{{ course.title }}"

{% endblock %}

{% block content %}

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

<div class="module">

<h2>Course modules</h2>

<form method="post">

{{ formset }}

{{ formset.management\_form }}

{% csrf\_token %}

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

</form>

</div>

{% endblock %}

Этот шаблон встраивает набор форм в элемент HTML <form> и добавляет управляющую форму для контроля набора форм.

Теперь перейдем к файлу list.html, где добавим ссылку на URL-адрес course\_module\_update под Edit и Delete link.

<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>

Таким образом, мы добавляем ссылку на редактирование модулей курса, давая возможность пользователям редактировать модули конкретного курса.

Наконец, после внесения этих изменений и сохранения файлов, мы можем протестировать результат, перейдя по URL-адресу <http://127.0.0.1:8000/course/mine/> в браузере. Создадим курс и кликнем по ссылке Edit modules (Редактировать модули). Рыс.14.

После этого мы увидим набор форм, включающий форму для каждого модуля, содержащегося в курсе, а также две пустые дополнительные формы, так как мы установили extra=2 для ModuleFormSet. **Рыс.15**.

**Добавление содержимого в модули курса**

Для начала нам необходим способ добавления содержимого – текста, видео, изображения и файлов – в модули курса. Мы могли бы создать четыре разных представления для каждого типа содержимого, но вместо этого мы предпочтем более универсальный подход. Мы создадим представление, которое обрабатывает создание или обновление объектов для любой модели содержимого.

Для этого мы будем редактировать файл views.py приложения courses, добавив в него следующий код:

from django.forms.models import modelform\_factory

from django.apps import apps

from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View):

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):

Form = modelform\_factory(model, exclude=['owner', 'order', 'created', 'updated'])

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

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

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

Это первая часть представления ContentCreateUpdateView. Оно позволяет создавать и обновлять содержимое разных моделей.

**Методы представления**

**get\_model()**

Этот метод проверяет принадлежность данного имени одной из четырех моделей содержимого: Text, Video, Image или File. Затем он использует модуль apps, встроенный в Django, чтобы получить фактический класс для данного имени модели. Если данное имя модели не является одним из допустимых, возвращается None.

**get\_form()**

Этот метод, используя функцию modelform\_factory() Django, создает динамическую форму. Поскольку форма будет компоноваться для моделей Text, Video, Image и File, здесь используется параметр exclude, чтобы указать общие поля, которые нужно исключить из формы, и разрешить автоматическое включение всех остальных атрибутов.

**dispatch()**

Этот метод получает параметры URL-адреса, сохраняет соответствующий модуль, модель и объект содержимого в качестве атрибутов класса:

* module\_id: ИД модуля, с которым ассоциировано или будет ассоциировано содержимое;
* model\_name: Имя модели содержимого, которое нужно создать или обновить;
* id: id обновляемого объекта. Он равен None, если создаются новые объекты.

Таким образом, с помощью этих методов мы можем динамически создавать и обновлять содержимое для различных моделей в нашем приложении.

**Добавление методов get() и post()**

Давайте начнем с методов get() и post(), которые мы добавляем в наше представление ContentCreateUpdateView.

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

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):

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:

# new content

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})

В методе get() мы формируем модельную форму для обновляемого экземпляра или для создания нового объекта, если instance не передан. И затем передаем эту форму для отображения вместе с объектом, который будет обновляться.

В методе post() мы сначала формируем модельную форму, передавая в неё данные и файлы, полученные из запроса. Затем, если форма валидна, создаем новый объект, назначаем текущего пользователя владельцем и сохраняем изменения. Если это новое содержимое, мы создаем объект Content и связываем его с модулем. Если форма не валидна, мы снова передаем форму для отображения вместе с объектом.

Далее, мы должны отредактировать наш файл urls.py, добавив новые шаблоны URL-адресов для управления созданием и обновлением содержимого модуля курса.

path('module/<int:module\_id>/content/<model\_name>/create/', views.ContentCreateUpdateView.as\_view(), name='module\_content\_create'),

path('module/<int:module\_id>/content/<model\_name>/<id>/', views.ContentCreateUpdateView.as\_view(), name='module\_content\_update'),

Теперь у нас есть два новых шаблона URL-адресов:

* module\_content\_create: используется для создания нового объекта содержимого и добавления его в модуль. Он содержит параметры module\_id и model\_name.
* module\_content\_update: используется для обновления существующего объекта содержимого. Он содержит параметры module\_id, model\_name и параметр id для идентификации обновляемого содержимого.

Наконец, мы создаем шаблон для представления ContentCreateUpdateView. Внутри каталога шаблонов courses/manage/ мы создадим новый каталог content.

courses/

manage/

content/

Затем в этом каталоге создадим файл form.html и добавим следующий код:

{% extends "base.html" %}

{% block title %}

{% if object %}

Edit content "{{ object.title }}"

{% else %}

Add new content

{% endif %}

{% endblock %}

{% block content %}

<h1>

{% if object %}

Edit content "{{ object.title }}"

{% else %}

Add 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 %}

Этот шаблон делает проверку на наличие переменной object в контексте. Если object существует, то мы обновляем существующий объект. В противном случае создаем новый объект.

Кроме того, в элемент формы <form> вставляется enctype="multipart/form-data, так как для моделей содержимого File и Image форма содержит загрузку файла на сайт.

Теперь мы можем динамически создавать и обновлять содержимое модуля курса, включая текст, видео, изображения и файлы, используя наши новые представления, URL-адреса и шаблоны HTML. **Рыс.16.**

Форма предоставляет возможность добавить новое изображение в курс.

Также, для удаления содержимого, нужно добавить представление. В файле views.py приложения courses предлагается добавить следующий код:

class ContentDeleteView(View):

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)

Этот класс извлекает объект Content с указанным id, удаляет связанный текст, видео, изображение или файл, и затем удаляет сам объект Content, перенаправляя пользователя на URL-адрес module\_content\_list для вывода списка другого содержимого модуля.

Теперь нам нужно обновить файл urls.py приложения courses с добавлением нового шаблона URL-адреса:

path('content/<int:id>/delete/', views.ContentDeleteView.as\_view(), name='module\_content\_delete'),

Теперь, преподаватели могут легко создавать, обновлять и удалять содержимое, делая процесс управления контентом более удобным и интуитивно понятным

**Управление модулями и их содержимым**

После создания представлений для создания, редактирования и удаления модулей курса и их содержимого, перейдем к созданию представлений отображения всех модулей курса и вывода списка содержимого конкретного модуля.

* Добавление представления и URL-адреса

В файле views.py добавим следующий код для создания представления ModuleContentListView:

class ModuleContentListView(TemplateResponseMixin, View):

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

def get(self, request, module\_id):

module = get\_object\_or\_404(Module, id=module\_id, course\_\_owner=request.user)

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

Это представление ModuleContentListView, которое получает объект Module с заданным id, принадлежащий текущему пользователю, и выводит шаблон с этим модулем.

Затем вносим изменения в файл urls.py, чтобы добавить новый шаблон URL-адреса:

path('module/<int:module\_id>/', views.ModuleContentListView.as\_view(), name='module\_content\_list'),

Теперь мы создали URL-адрес module\_content\_list, который будет связан с представлением ModuleContentListView.

* Создание шаблона content\_list.html

Следующим шагом, создадим новый шаблон под названием content\_list.html в каталоге templates/courses/manage/module/. В этот шаблон добавим 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 %}>

<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 " 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 " 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 отвечает за отображение модуля и содержимого внутри него. Мы подставляем название курса, затем отображаем список модулей курса и содержимого конкретного модуля.

Таким образом, мы расширяем функциональность платформы, делая управление модулями и их содержимым более удобным и интуитивным.

**Обновление шаблона для отображения модулей и содержимого**

Наша задача — обновить шаблон, отображающий все модули курса и содержимое выбранного модуля. Мы будем использовать циклы для отображения модулей курса и их содержимого.

* Создание шаблонного фильтра

Первым делом, нам нужен способ определить тип каждого объекта содержимого: Text, Video, Image или File. Для этого мы можем написать шаблонный фильтр.

Сначала создадим файловую структуру в каталоге приложения courses:

templatetags/

\_\_init\_\_.py

course.py

Затем отредактируем модуль course.py, добавив в него следующий шаблонный фильтр model\_name:

from django import template

register = template.Library()

@register.filter

def model\_name(obj):

try:

return obj.\_meta.model\_name

except AttributeError:

return None

Этот шаблонный фильтр model\_name позволит нам получать модельное имя для объекта в шаблонах, используя выражение {{ object|model\_name }}.

* Обновление шаблонов

Затем мы обновим шаблон content\_list.html. Мы будем использовать наш созданный шаблонный фильтр.

Первым делом подключим наш шаблонный фильтр в шаблоне content\_list.html:

{% extends "base.html" %}

{% load course %} #new code

{% block title %}

####

Затем найдем строки, где отображается содержимое модуля

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

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

и заменим их следующим образом:

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

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

Используя выражение {{ item|model\_name }}, мы отображаем модельное имя элемента содержимого в шаблоне, а также используем его для формирования ссылки на редактирование объекта.

* Добавление ссылки на управление содержимым

Далее, мы добавим ссылку на управление содержимым в шаблоне course/list.html:

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

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

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

Manage contents

</a>

{% endif %}

Эта новая ссылка позволяет пользователям обратиться к содержимому первого модуля курса, если таковые имеются.

После этих изменений, остановим сервер разработки и затем снова его запустим с помощью команды: python manage.py runserver

Пройдем по URL-адресу <http://127.0.0.1:8000/course/mine/> и кликнем по ссылке Manage contents (Управление содержимым) на курс, который содержит хотя бы один модуль. Вы увидите страницу, похожую на приведенную в **Рыс.17**.

Здесь, при нажатии на модуль, его содержимое будет отображено в главной области. Также шаблон содержит ссылки на добавление нового текста, видео, изображения или файла для отображаемого модуля.

Добавьте в модуль пару разных типов содержимого и посмотрите на результат. Содержимое модуля появится под надписью Module contents (Содержимое модуля), как показано на **Рыс.18**.

**Переупорядочивание модулей и их содержимого**

**Использование примесей из модуля django-braces**

У нас будут эти два класса-примеси:

* CsrfExemptMixin: этот парень помогает избежать проверки CSRF токена в POST-запросах. А это значит, что если ты делаешь AJAX POST-запросы, тебе не нужно беспокоиться о передаче csrf токена.
* JsonRequestResponseMixin: он занимается преобразованием данных запроса в JSON и также сериализует ответ в JSON, отправляя обратно HTTP-ответ с заголовком application/json.

Для начала дела, нам нужно установить этот модуль через pip командой **pip install django-braces**.

Теперь в приложении тебе нужно обновить порядок модулей курса в формате JSON. Для этого будем использовать вот такой класс в файле views.py приложения courses:

from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class ModuleOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):

def post(self, request):

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 представление ModuleOrderView дает возможность обновлять порядок модулей курса без лишних хлопот.

Не будем ограничиваться только порядком модулей! Можно также упорядочить содержимое модуля. Давай просто добавим подобное представление:

class ContentOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):

def post(self, request):

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'})

После этого не забудь обновить urls.py в своем приложении, чтобы указать пути для этих представлений:

path('module/order/', views.ModuleOrderView.as\_view(), name='module\_order'),

path('content/order/', views.ContentOrderView.as\_view(), name='content\_order'),

И затем, чтобы добавить функциональность перетаскивания в своем шаблоне, можно воспользоваться библиотекой HTML5 Sortable, которая позволяет удобно сортировать элементы с помощью функционала Drag and Drop в HTML.

Ну и в конце концов, чтобы вставлять JavaScript файлы в любой шаблон, расширяющий base.html, можешь использовать такой вот код:

{% load static %}

<!DOCTYPE html>

<html>

<head>

# ...

</head>

<body>

<div id="header">

# ...

</div>

<div id="content">

{% block content %}

{% endblock %}

</div>

###new code

{% block include\_js %}

{% endblock %}

###

<script>

document.addEventListener('DOMContentLoaded', (event) => {

// DOM-модель загружена

{% block domready %}

{% endblock %}

})

</script>

</body>

</html>

И вот теперь, благодаря этому новому блоку include\_js, ты можешь подключать JavaScript файлы в любой шаблон, который расширяет base.html.

Давай немного изменим шаблон courses/manage/module/content\_list.html и придадим ему немного драйва с помощью JavaScript!

Вот что мы делаем: в самый конец этого шаблона мы добавляем этот кусочек кода, который как раз и загружает библиотеку HTML5 Sortable из общедоступной сети доставки контента (CDN):

{% block content %}

# ...

{% endblock %}

###new code

{% block include\_js %}

<script src="https://cdnjs.cloudflare.com/ajax/libs/html5sortable/0.13.3/html5sortable.min.js"></script>

{% endblock %}

Этот код добавляет крутую функциональность сортировки элементов на странице. Загружаем эту библиотеку из Интернета, чтобы не нагружать наш сервер. Уже в предыдущей главе мы использовали CDN для загрузки JavaScript библиотек, и это снова оказывается весьма удобным.

А вот далее вставляем еще кусочек JavaScript-кода в блок domready в самом низу:

{% block domready %}

var options = { method: 'POST', mode: 'same-origin' };

const moduleOrderUrl = '{% url "module\_order" %}';

{% endblock %}

Этот новый код JavaScript определен в блоке {% block domready %} и будет выполнен после загрузки DOM страницы. Здесь мы подготавливаем опции для HTTP-запроса, который будет использоваться для переупорядочивания модулей. Мы создаем URL для запроса и сохраняем его для дальнейшего использования.

Далее давай добавим еще кусочек кода в блок domready, который снова добавит еще больше функциональности:

{% block domready %}

var options = {

method: 'POST',

mode: 'same-origin'

};

const moduleOrderUrl = '{% url "module\_order" %}';

sortable('#modules', {

forcePlaceholderSize: true,

placeholderClass: 'placeholder'

});

{% endblock %}

тот новый код JavaScript определяет, что наш HTML-элемент с id="modules" станет сортируемым благодаря библиотеке HTML5 Sortable. При перетаскивании элементов будет появляться заменитель, чтобы было понятно, куда элементы будут перемещены. Мы также устанавливаем параметр forcePlaceholderSize в true, чтобы заменитель имел высоту, и указываем класс placeholder, который будет стилизовать этот заменитель. **Рыс.19**.

И теперь, давай дополним нашу крутую функциональность сортировки еще дальше — дадим преподавателям возможность управлять порядком содержимого модулей.

Вот что мы делаем: к нашему блоку domready в шаблоне courses/manage/module/content\_list.html добавляем следующий код:

{% block domready %}

// ...

const contentOrderUrl = '{% url "content\_order" %}';

sortable('#module-contents', {

forcePlaceholderSize: true,

placeholderClass: 'placeholder'

})[0].addEventListener('sortupdate', function(e) {

contentOrder = {};

var contents = document.querySelectorAll('#module-contents div');

contents.forEach(function (content, index) {

contentOrder[content.dataset.id] = index;

options['body'] = JSON.stringify(contentOrder);

fetch(contentOrderUrl, options);

});

});

{% endblock %}

Тут вместо module\_order мы используем content\_order, а функциональность перетаскивания применяется к HTML-элементу с идентификатором module-contents. Подход тот же самый, что и для сортировки модулей курса. Отличие только в том, что здесь нет необходимости обновлять нумерацию содержимого, так как у него нет видимого порядкового номера.

**Прорисовка и кеширование**

**Отображение курсов**

Мы должны добавить несколько ключевых функциональностей:

1. Вывод списка всех курсов, возможно отфильтрованных по предмету.
2. Отображение краткого обзора одного курса.

Для этого редактируем файл views.py в приложении courses, добавляя следующий код:

from .models import Subject

class CourseListView(TemplateResponseMixin, View):

model = Course

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

def get(self, request, subject=None):

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})

В представлении CourseListView мы делаем следующее:

1. Используем метод annotate() ORM и агрегирующую функцию Count(), чтобы извлечь все предметы с общим числом курсов для каждого предмета.
2. Извлекаем все курсы, включая общее количество модулей в каждом курсе.
3. Если указан параметр URL subject, мы извлекаем соответствующий объект предмета и фильтруем курсы по этому предмету.
4. Для рендеринга объектов по шаблону и возврата HTTP-ответа используется метод render\_to\_response().

Теперь мы добавим представление для отображения подробной информации о курсе, чтобы показать краткий обзор одного курса:

from django.views.generic.detail import DetailView

class CourseDetailView(DetailView):

model = Course

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

В представлении CourseDetailView мы указываем модель и шаблон для отображения деталей курса.

Далее, в главном файле urls.py проекта educa, добавляем следующий шаблон URL-адреса:

from courses.views import CourseListView

urlpatterns = [

# ...

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

]

Шаблон URL-адреса course\_list добавляется в главный файл URL-адресов проекта, поскольку по адресу <http://127.0.0.1:8000/> планируется показывать список курсов.

Далее, в файле urls.py приложения courses, добавляем шаблоны URL-адресов для отображения списка курсов по предмету и краткого обзора одного курса:

path('subject/<slug:subject>/', views.CourseListView.as\_view(), name='course\_list\_subject'),

path('<slug:slug>/', views.CourseDetailView.as\_view(), name='course\_detail'),

Здесь определены следующие шаблоны URL-адресов:

* course\_list\_subject: для отображения всех курсов по предмету.
* course\_detail: для отображения краткого обзора одного курса.

Давай создадим стильные шаблоны представлений CourseListView и CourseDetailView!

Для начала, внутри каталога templates/courses/ в приложении courses, создадим следующую файловую структуру:

course/

list.html

detail.html

Далее, внесем изменения в шаблон courses/course/list.html:

{% 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 "course\_list\_subject" s.slug %}">

{{ s.title }} <br>

<span>

{{ s.total\_courses }} course{{ s.total\_courses|pluralize }}

</span>

</a>

</li>

{% endfor %}

</ul>

</div>

<div class="module">

{% for course in courses %}

{% with subject=course.subject %}

<h3>

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

{{ course.title }}

</a>

</h3>

<p>

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

{{ course.total\_modules }} modules.

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

</p>

{% endwith %}

{% endfor %}

</div>

{% endblock %}

Этот шаблон списка курсов отображает красивый HTML-список всех предметов и их курсов, а также общее количество модулей и информацию о преподавателе для каждого курса.

Далее, отредактируем шаблон courses/course/detail.html:

{% extends "base.html" %}

{% block title %}

{{ object.title }}

{% endblock %}

{% block content %}

{% with subject=object.subject %}

<h1>

{{ object.title }}

</h1>

<div class="module">

<h2>Overview</h2>

<p>

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

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

{{ object.modules.count }} modules.

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

</p>

{{ object.overview|linebreaks }}

</div>

{% endwith %}

{% endblock %}

Этот шаблон детализирует краткий обзор и дополнительную информацию о выбранном курсе, включая общее количество модулей, информацию о преподавателе и обзор курса.

После внесения всех этих изменений, взгляни на великолепный результат, пройдя по URL-адресу <http://127.0.0.1:8000/> в своем браузере. Ты увидишь шикарную страницу списка курсов, где в левой боковой панели перечислены все предметы с общим числом курсов по каждому. **Рыс.20.**

Теперь, когда ты кликнешь по одному из курсов, откроется страница краткого обзора курса со всеми необходимыми деталями.

Затем выберите один из курсов для просмотра. После этого отобразится страница с указанной структурой **Рыс.21**.

**Добавление регистрации студентов**

**Создание функционала для регистрации студентов**

Для создания нового приложения выполните следующую команду **python manage.py startapp students.**

После этого отредактируем файл settings.py вашего проекта education, добавив новое приложение в параметр INSTALLED\_APPS:

NSTALLED\_APPS = [

# ...

'students.apps.StudentsConfig',

]

**Создание представления регистрации студентов в Django**

Сначала мы отредактируем файл views.py в приложении students. В нем есть класс StudentRegistrationView, который использует Django представление CreateView для создания новых пользователей.

class StudentRegistrationView(CreateView):

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

form\_class = UserCreationForm

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

def form\_valid(self, form):

result = super().form\_valid(form)

cd = form.cleaned\_data

user = authenticate(username=cd['username'], password=cd['password1'])

login(self.request, user)

return result

Здесь мы указываем следующие атрибуты:

* template\_name: путь к файлу шаблона, который используется для отображения этого представления;
* form\_class: класс формы для создания объектов, в данном случае, для регистрации пользователя используется встроенная форма UserCreationForm;
* success\_url: URL, на который перенаправляется пользователь после успешной регистрации. В данном случае, после успешной регистрации пользователь будет перенаправлен на страницу списка курсов.

Метод form\_valid() вызывается при отправке валидной формы. Он должен вернуть HTTP-ответ. Мы переопределяем этот метод, чтобы после успешной регистрации пользователя автоматически выполнялся вход на сайт.

Затем мы создаем файл urls.py внутри каталога приложения students и добавляем следующий код:

from django.urls import path

from . import views

urlpatterns = [

path('register/', views.StudentRegistrationView.as\_view(), name='student\_registration'),

]

После этого отредактируем основной файл urls.py проекта education, добавив путь к приложению students:

urlpatterns = [

# ...

path('students/', include('students.urls')),

]

Для шаблонов, мы создаем следующую файловую структуру внутри каталога приложения students:

- templates/

- students/

- student/

- registration.html

И редактируем сам шаблон registration.html, добавляя следующий код:

{% 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 method="post">

{{ form.as\_p }}

{% csrf\_token %}

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

</form>

</div>

{% endblock %}

После всех этих шагов, если запустить сервер разработки и перейти по URL-адресу <http://127.0.0.1:8000/students/register/>, появится регистрационная форма на сайте.

**Создание функциональности зачисления студентов на курсы**

Сначала мы отредактируем файл models.py в приложении courses, добавив поле students в модель Course, чтобы установить связь многие-ко-многим между курсами и пользователями.

from django.contrib.auth.models import User

from django.db import models

class Course(models.Model):

# Другие поля модели

students = models.ManyToManyField(User, related\_name='courses\_joined', blank=True)

После этого, мы создадим миграцию и применим новые миграции, что позволяет нам обновить базу данных соответствующим образом.

Теперь, когда у нас есть связь между студентами и курсами, давайте создадим функциональность зачисления студентов на курсы.

Сначала создадим файл forms.py внутри каталога приложения students и добавим в него форму CourseEnrollForm.

from django import forms

from courses.models import Course

class CourseEnrollForm(forms.Form):

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

Эта форма CourseEnrollForm используется для зачисления студентов на курсы. Поле course представляет выбранный курс, а тип поля ModelChoiceField используется для выбора курса. Также мы используем HiddenInput виджет, чтобы скрыть это поле от пользователя.

Далее, отредактируем файл views.py в приложении students, добавив класс StudentEnrollCourseView, который обрабатывает зачисление студентов на курсы.

from django.views.generic.edit import FormView

from django.contrib.auth.mixins import LoginRequiredMixin

from .forms import CourseEnrollForm

class StudentEnrollCourseView(LoginRequiredMixin, FormView):

course = None

form\_class = CourseEnrollForm

def form\_valid(self, form):

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

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

return super().form\_valid(form)

def get\_success\_url(self):

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

В данном классе StudentEnrollCourseView, мы используем LoginRequiredMixin для защиты доступа к представлению только залогиненных пользователей. Мы также используем FormView для обработки формы. После успешной валидации формы, текущий пользователь добавляется к студентам, зачисленным на выбранный курс.

Метод get\_success\_url() возвращает URL-адрес, на который пользователя перенаправит после успешной зачисления на курс. Атрибут success\_url эквивалентен методу get\_success\_url().

Добавим новый URL-маршрут в файл urls.py приложения students:

path('enroll-course/', views.StudentEnrollCourseView.as\_view(), name='student\_enroll\_course'),

Этот URL будет направлять наше представление StudentEnrollCourseView, которое обрабатывает зачисление студента на курс.

Теперь отредактируем класс CourseDetailView в файле views.py приложения courses, чтобы включить форму зачисления на курс:

from students.forms import CourseEnrollForm

class CourseDetailView(DetailView):

model = Course

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

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

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

context['enroll\_form'] = CourseEnrollForm(initial={'course': self.object})

return context

Метод get\_context\_data() используется для включения формы зачисления в контекст отображения шаблона. Поле course формы инициализируется текущим объектом Course, чтобы передать его для обработки.

В файле шаблона courses/course/detail.html, мы заменяем определенную часть кода

{{ object.overview|linebreaks }}

на следующий образ:

{{ object.overview|linebreaks }}

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

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

{{ enroll\_form }}

{% csrf\_token %}

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

</form>

{% else %}

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

Register to enroll

</a>

{% endif %}

Этот код добавляет кнопку зачисления на курс на страницу краткого обзора курса. Если пользователь аутентифицирован, будет отображена кнопка с формой зачисления, отправляющая данные на URL student\_enroll\_course. Если пользователь не аутентифицирован, будет показана ссылка для регистрации на платформе.

Теперь если перейти по URL-адресу <http://127.0.0.1:8000/> и щелкнуть по курсу ты должен увидеть кнопку "ENROLL NOW" (Записаться сейчас) под кратким обзором курса. **Рыс.22**.

А если ты не вошел на платформу, тогда увидишь кнопку "REGISTER TO ENROLL" (Зарегистрироваться, чтобы записаться) вместо нее. **Рыс.23**.

**Доступ к содержимому курсов**

Представление StudentCourseListView

from django.views.generic.list import ListView

from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView):

model = Course

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

def get\_queryset(self):

qs = super().get\_queryset()

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

Это представление отображает курсы, на которые зачислены студенты. Оно использует класс LoginRequiredMixin для обеспечения доступа только залогиненным пользователям. Также используется ListView для отображения списка объектов Course. Метод get\_queryset() переопределен для извлечения курсов, на которые зачислен конкретный студент.

Представление StudentCourseDetailView:

from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):

model = Course

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

def get\_queryset(self):

qs = super().get\_queryset()

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

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

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

course = self.get\_object()

if 'module\_id' in self.kwargs:

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

else:

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

return context

Это представление позволяет студентам просматривать фактическое содержимое курсов. Метод get\_queryset() ограничивает доступ к курсам, на которые студент зачислен. Метод get\_context\_data() добавляет модуль курса в контекст, если указан параметр module\_id в URL.

URL-адреса в файле urls.py:

path('courses/', views.StudentCourseListView.as\_view(), name='student\_course\_list'),

path('course/<pk>/', views.StudentCourseDetailView.as\_view(), name='student\_course\_detail'),

path('course/<pk>/<module\_id>/', views.StudentCourseDetailView.as\_view(), name='student\_course\_detail\_module'),

Эти URL-адреса определяют доступ студентов к списку курсов, деталям курса и модулям курса соответственно.

Шаблон student/course/list.html:

{% 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 "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 on a course.

</p>

{% endfor %}

</div>

{% endblock %}

Этот шаблон отображает курсы, на которые зачислен студент, и предоставляет доступ к содержимому курсов через ссылку "Access contents".

Внесем несколько изменений в файл настроек settings.py нашего проекта education, чтобы задать поведение перенаправления после успешного входа на платформу:

from django.urls import reverse\_lazy

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

Этот параметр определяет адрес, на который студент будет перенаправлен после успешного входа на сайт, если не указан параметр next в запросе. После входа студент будет автоматически перенаправлен на страницу student\_course\_list для просмотра курсов, на которые он зачислен.

Теперь внимание на шаблон student/course/detail.html, в который мы добавим следующий код:

{% 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>

</span>

<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 показа содержимого курса студентам, которые зачислены на него. Сначала создается HTML-список всех модулей курса, выделяя текущий модуль. Затем содержимое текущего модуля прорисовывается в цикле, используя {{ item.render }}, чтобы отобразить каждый элемент содержимого. Для моделей содержимого добавляется метод render(), который обеспечивает корректное отображение содержимого.

Теперь, после регистрации нового студента на адресе <http://127.0.0.1:8000/students/register/>, он сможет записаться на курс и просматривать содержимое курса.

**Прорисовка разных типов содержимого**

Чтобы волшебно показать содержимое курса, нужно нарисовать все ваши типы контента: текст, изображения, видео и файлы. Давайте добавим немного магии в файл models.py вашего приложения courses, внедрив этот следующий метод render() в модель ItemBase:

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

class ItemBase(models.Model):

# ...

def render(self):

return render\_to\_string(

f'courses/content/{self.\_meta.model\_name}.html',

{'item': self}

)

В этом волшебном методе используется функция render\_to\_string(), которая по-волшебному отображает шаблон и возвращает его в виде вдохновляющей строки. Каждый тип содержимого будто рисуется своим собственным шаблоном, по имени модели содержимого. Параметр self.\_meta.model\_name добавляется для того, чтобы динамически создавать нужное имя шаблона для каждого типа содержимого. 🌟

Теперь отправимся в магический мир шаблонов! В папке templates/courses/ вашего приложения courses, создадим вот такую магическую структуру файлов:

content/

text.html

file.html

image.html

video.html

В шаблон text.html напишем следующий магический код:

{{ item.content|linebreaks }}

Этот шаблон создан для отображения текстового контента. Фильтр linebreaks как по волшебству заменяет разрывы строк в обычном тексте на HTML-разрывы строк.

Редактируем шаблон file.html и добавляем в него следующее:

<p>

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

</p>

Этот шаблон создает волшебную ссылку для загрузки файла.

Далее открываем шаблон image.html и волшебным образом записываем:

<p>

<img src="{{ item.file.url }}" alt="{{ item.title }}">

</p>

Этот шаблон как бы волшебно отображает изображения.

Прежде всего, установим **pip install django-embed-video**  
Отредактируем файл settings.py вашего проекта, добавив туда это волшебное приложение:

INSTALLED\_APPS = [

# ...

'embed\_video',

]

Теперь открываем шаблон video.html и вписываем этот волшебный код:

{% load embed\_video\_tags %}

{% video item.url "small" %}

И вот — шаблон для воспроизведения видео. **Рыс 24**.

**Использование кеш-фреймворка**

**Использование кеш-бэкенда Redis**

C Django 4.0 был добавлен кеш-бэкенд Redis, который мы можем использовать вместо Memcached в нашем проекте. Давайте внесем необходимые изменения в настройки для использования Redis в качестве кеш-бэкенда.

Сначала установим библиотеку redis-py с помощью следующей команды **pip install redis**.

Затем отредактируем файл settings.py нашего проекта education, изменив параметр CACHES следующим образом:

CACHES = {

'default': {'BACKEND': 'django.core.cache.backends.redis.RedisCache','LOCATION': 'redis://127.0.0.1:6379',}}

Теперь наш проект будет использовать кеш-бэкенд RedisCache. Мы указываем местоположение Redis в формате redis://[хост]:[порт], где мы используем адрес 127.0.0.1 для локального хоста и порт 6379, который является портом по умолчанию для Redis.

Далее инициализируем контейнер Redis на платформе Docker с помощью следующей команды **docker run -it --rm --name redis -p 6379:6379 redis**

**Отслеживание сервера Redis с помощью**

Для начала установим пакет django-redisboard, который позволит нам контролировать наш сервер Redis через административный интерфейс Django. С помощью команды **pip install django-redisboard** мы устанавливаем необходимый пакет.

Для корректной работы приложения Django Redisboard также установим библиотеку Python attrs с помощью команды **pip install attrs**.

Теперь отредактируем файл settings.py нашего проекта, добавив приложение redisboard в параметр INSTALLED\_APPS. Это позволит Django использовать функционал Django Redisboard.

После этого выполним миграцию приложения Django Redisboard с помощью команды **python manage.py migrate redisboard** чтобы подготовить базу данных к работе с новым функционалом.

Запустим сервер разработки и перейдем по URL-адресу<http://127.0.0.1:8000/admin/redisboard/redisserver/add/> в браузере, чтобы добавить сервер Redis и начать его отслеживать. Заполним поля Label (Метка) и URL, указав соответственно название и адрес нашего сервера Redis (например, redis://localhost:6379/0). **Рыс.25**

После сохранения информации в базе данных мы сможем просматривать конфигурацию и метрики сервера Redis через административный интерфейс Django Redisboard. **Рыс.26**.

**Разработка API**

**Django REST framework**

Django REST framework (DRF) - это фреймворк для создания RESTful API в Django. Он следует принципам REST, где ресурсы представляются моделями, а HTTP-методы используются для работы с объектами. DRF упрощает разработку API, предоставляя возможности извлечения элементов, курсов и содержимого курса, а также записи на курс.

DRF поддерживает форматы JSON и XML для обмена данными. Установка DRF осуществляется через pip, а затем его необходимо добавить в INSTALLED\_APPS в settings.py. Настройки DRF позволяют определить права доступа, например, DjangoModelPermissionsOrAnonReadOnly, которые позволяют пользователям выполнять различные действия над объектами.

**Исследование Сериализаторов в Django REST framework**

После установки DRF необходимо создать сериализаторы для правильного преобразования данных. Фреймворк предоставляет классы сериализаторов, такие как Serializer, ModelSerializer и HyperlinkedModelSerializer для работы с объектами.

Для создания первого сериализатора создайте файл serializers.py в каталоге api внутри каталога courses. В этом файле определите сериализатор SubjectSerializer для модели Subject, указав необходимые поля для сериализации.

from courses.models import Subject

from courses.api.serializers import SubjectSerializer

subject = Subject.objects.latest('id')

serializer = SubjectSerializer(subject)

serializer.data

{'id': 5, 'title': 'Internet Marketer', 'slug': 'internet-marketer'}

Этот код позволяет получить объект Subject, создать экземпляр SubjectSerializer и извлечь сериализованные данные.

**Погружение в Мир Парсеров и Рендереров**

В Django REST framework есть парсеры (для разбора) и рендереры (для форматирования) данных. Парсеры используются для разбора входящих данных, а рендереры для форматирования ответов API.

Пример использования JSONParser для разбора данных из JSON в объект Python:

from io import BytesIO

from rest\_framework.parsers import JSONParser

data = b'{"id": 5, "title": "Internet Marketer", "slug": "internet-marketer"}'

JSONParser().parse(BytesIO(data))

{'id': 5, 'title': 'Internet Marketer', 'slug': 'internet-marketer'}

Также Django REST framework предоставляет рендереры, которые выбираются на основе заголовка Accept запроса или форматного суффикса URL-адреса. Например, JSONRenderer используется для преобразования сериализованных данных в формат JSON:

from rest\_framework.renderers import JSONRenderer

JSONRenderer().render(serializer.data)

По умолчанию DRF использует JSONRenderer и BrowsableAPIRenderer для удобного просмотра API через веб-интерфейс.

**Создание представлений списка и детальной информации в Django REST Framework**

Это удобный способ работы с моделями данных. Мы можем легко извлекать, создавать, обновлять и удалять объекты с помощью представлений.

Для создания представлений списка и детальной информации объектов Subject в Django REST Framework, нужно создать файл views.py в директории courses/api/ и добавить следующий код:

from rest\_framework import generics

from courses.models import Subject

from courses.api.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

Эти представления используют типовые классы ListAPIView и RetrieveAPIView из Django REST Framework. В представлении детальной информации используется URL-параметр pk для извлечения объекта по его первичному ключу.

Оба представления имеют два важных атрибута:

- queryset: базовый QuerySet для извлечения объектов;

- serializer\_class: класс сериализатора для объектов.

Для добавления URL-адресов для этих представлений, нужно создать файл urls.py в courses/api/ и добавить следующий код:

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'),

]

Основной urls.py в проекте education:

urlpatterns = [

# ...

path('api/', include('courses.api.urls', namespace='api')),

]

**Потенциал сериализации в Django**

**Создание сериализатора для модели Course:**

Я создаю сериализатор для модели Course. Для этого в файле api/serializers.py моего приложения courses я редактирую код:

from courses.models import Subject, Course

from rest\_framework import serializers

class CourseSerializer(serializers.ModelSerializer):

class Meta:

model = Course

fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']

Запускаю оболочку Django с помощью python manage.py shell и выполняю код:

from rest\_framework.renderers import JSONRenderer

from courses.models import Course

from courses.api.serializers import CourseSerializer

course = Course.objects.latest('id')

serializer = CourseSerializer(course)

JSONRenderer().render(serializer.data)

Если хочу получить более подробную информацию о модулях, мне нужно вложить их. Для этого вношу изменения в сериализатор CourseSerializer, добавляя код в api/serializers.py:

from rest\_framework import serializers

from courses.models import Subject, Course, Module

class ModuleSerializer(serializers.ModelSerializer):

class Meta:

model = Module

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

class CourseSerializer(serializers.ModelSerializer):

modules = ModuleSerializer(many=True, read\_only=True)

class Meta:

model = Course

fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']

В новом коде мною определен ModuleSerializer для сериализации модели Module, и добавлен атрибут modules в CourseSerializer для вложения сериализатора ModuleSerializer.

Тестирую изменения, создavaя экземпляр CourseSerializer и просматривая JSON-представление атрибута data сериализатора с помощью JSONRenderer. Теперь модули сериализуются, используя вложенный сериализатор ModuleSerializer.

"modules": [

{"order": 0,

"title": "Hello World",

"description": "Python is a programming language.Like other..."},

{"order": 1,

"title": "Control Flow",

"description": "Imagine waking up in the morning..."}

]

**Разработка конкретно-прикладных представлений API в Django**

Класс APIView в Django позволяет создавать API-функционал поверх класса View, встроенного в Django, но с использованием объектов Request и Response от REST framework. Также в нем есть обработка исключений типа APIException для правильных HTTP-ответов. К тому же не забудем про встроенную систему аутентификации и авторизации для контроля доступа к нашим представлениям.

Я хочу создать представление, чтобы зачислить пользователей на курсы. Для этого я редактирую файл api/views.py в моем приложении courses следующим образом:

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

from rest\_framework.views import APIView

from rest\_framework.response import Response

from courses.models import Subject, Course

from courses.api.serializers import SubjectSerializer

class CourseEnrollView(APIView):

def post(self, request, pk, format=None):

course = get\_object\_or\_404(Course, pk=pk)

course.students.add(request.user)

return Response({'enrolled': True})

Разбор представления CourseEnrollView:

- Создал конкретно-прикладное представление как подкласс APIView.

- Определил метод post() для обработки действий типа POST, который единственный разрешенный HTTP-метод для этого представления.

- Ожидаю URL-параметр pk с ID курса. Если курс не найден, вызывается исключение 404.

- Текущий пользователь добавляется к студентам курса (многие ко многим), и возвращается успешный ответ.

Для доступа к этому представлению по URL, добавляю шаблон URL-адреса для CourseEnrollView в файл api/urls.py:

path('courses/<pk>/enroll/', views.CourseEnrollView.as\_view(), name='course\_enroll'),

Теперь я могу отправить POST-запрос, чтобы зачислить текущего пользователя на курс.

**Мощь Наборов Представлений (ViewSet) и Маршрутизаторов в REST API**

Использование ViewSet помогает избежать повторения логики в разных представлениях, предоставляя стандартные операции:

- Создание: create()

- Извлечение: list() и retrieve()

- Обновление: update() и partial\_update()

- Удаление: destroy()

Для модели Course я создаю ViewSet. В файле api/views.py добавляю следующий код:

from rest\_framework import viewsets

from courses.api.serializers import CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

Этот код создает подкласс ReadOnlyModelViewSet, который позволяет только операции на чтение list() и retrieve() для списка объектов и извлечения одиночного объекта.

Для создания маршрута для нашего ViewSet я редактирую файл api/urls.py:

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)),

]

Здесь мы создаем объект DefaultRouter и регистрируем ViewSet с префиксом "courses". Маршрутизатор автоматически генерирует URL-адреса для набора представлений ViewSet.

Теперь, если открыть браузер и перейти по адресу http://127.0.0.1:8000/api/, вы увидите список всех ViewSet по базовому URL-адресу.

По адресу http://127.0.0.1:8000/api/courses/ будет отображен список курсов.

**Расширение Возможностей с Дополнительными Действиями в Наборах Представлений**

**Добавление действия enroll в CourseViewSet**

Для этого я преобразую уже знакомое представление CourseEnrollView в специальное действие набора представлений CourseViewSet в файле api/views.py:

from rest\_framework.decorators import action

class CourseViewSet(viewsets.ReadOnlyModelViewSet):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

@action(detail=True, 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})

Обновляю файл api/urls.py, удаляя или комментируя старый URL для courses/<pk>/enroll/, так как теперь он не нужен. Затем в файле api/views.py удаляю или комментирую класс CourseEnrollView, так как URL для зачисления студентов теперь создается автоматически маршрутизатором.

Хочу предоставить доступ к содержимому курса только зачисленным студентам. Для этого создаю конкретно-прикладной класс разрешения в файле courses/api/permissions.py:

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()

Вот таким образом я расширяю возможности API с дополнительными действиями в наборах представлений.

**Сериализация содержимого курса с Django REST framework**

У в модели Content есть типичный внешний ключ для связи объектов разных моделей контента в приложении. Как сериализовать содержимое курса, чтобы передавать его через API.

1. Сериализация: Превращение данных в формат, удобный для передачи и хранения. В данном случае, нужно преобразовать информацию о курсе в формат для отправки через API.

2. Код: Класс ItemRelatedField наследуется от RelatedField из Django REST framework. Метод to\_representation возвращает прорисованное значение для связанных объектов.

from rest\_framework import serializers

from courses.models import Subject, Course, Module, 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

fields = ['order', 'item']

3. Значение: В ContentSerializer определяешь, какие данные из модели Content будут отправлены через API. ItemRelatedField помогает преобразовывать связанные объекты для представления.

Расширение функциональности сериализации:

- Созданы сериализаторы ModuleWithContentsSerializer и CourseWithContentsSerializer для включения содержимого модуля.

- Добавлен метод contents в CourseViewSet, чтобы получить содержимое курса через API.

Выглядит это вот так:

from rest\_framework import serializers

from courses.api.serializers import ContentSerializer

class ModuleWithContentsSerializer(serializers.ModelSerializer):

contents = ContentSerializer(many=True)

class Meta:

model = Module

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

class CourseWithContentsSerializer(serializers.ModelSerializer):

modules = ModuleWithContentsSerializer(many=True)

class Meta:

model = Course

fields = ['id', 'subject', 'title', 'slug', 'overview', 'created', 'owner', 'modules']

Метод contents в CourseViewSet позволяет получить содержимое курса через API только для зачисленных пользователей. При обращении по URL вы увидишь прорисованное HTML содержимое каждого модуля.

**Потребление RESTful API с использованием Python и библиотеки requests:**

При использовании готового API, можно интегрировать его в различные приложения для взаимодействия. Библиотека requests в Python помогает взаимодействовать с API, отправляя HTTP-запросы.

Пример потребления API с requests:

import requests

base\_url = 'http://127.0.0.1:8000/api/'

r = requests.get(f'{base\_url}courses/')

courses = r.json()

available\_courses = ', '.join([course['title'] for course in courses])

print(f'Available courses: {available\_courses}')

Этот код отправляет GET-запрос к API для получения данных о курсах. Полученные курсы выводятся на экран.

Обновленный скрипт с зачислением:

import requests

username = 'your\_username'

password = 'your\_password'

base\_url = 'http://127.0.0.1:8000/api/'

r = requests.get(f'{base\_url}courses/')

courses = r.json()

available\_courses = ', '.join([course['title'] for course in courses])

print(f'Available courses: {available\_courses}')

for course in courses:

course\_id = course['id']

course\_title = course['title']

r = requests.post(f'{base\_url}courses/{course\_id}/enroll/', auth=(username, password))

if r.status\_code == 200:

print(f'Successfully enrolled in {course\_title}')

Этот скрипт:

- Получает доступные курсы через API.

- Заходит на каждый курс и зачисляет пользователя по одному.

- По завершении, выводится список доступных курсов и информация о зачислении.

Таким образом, при использовании библиотеки requests в Python, можно взаимодействовать с API, отправляя запросы и обрабатывая ответы. Это простой и эффективный способ потребления API в своих приложениях.

**Разработка чат-сервера**

**Разработка чат-приложения с Django Channels**

Создадим чат-сервер для студентов, чтобы они могли общаться в реальном времени в чат-комнатах по каждому курсу, используя Django Channels, которые поддерживают протоколы с длительными соединениями, такие как веб-сокеты.

1. Использование Django Channels:

Django Channels позволяет легко добавить реально-временные и асинхронные функции вместе со стандартными HTTP-представлениями. Создаем новое приложение с именем chat, чтобы реализовать логику чат-сервера.

2. Шаги по созданию чат-сервера:

- Создаем приложение chat в проекте с помощью django-admin startapp chat.

- Активируем это приложение в настройках проекта settings.py, добавив 'chat' в список INSTALLED\_APPS.

Теперь у нас активировано приложение chat в проекте и готово для дальнейшего развития чат-сервера.

**Создание представления и шаблона для чат-комнаты с использованием Django**

Пошагово добавим возможность студентам общаться в чат-комнатах для каждого курса с помощью Django.

1. Разработка представления:

- Создаем представление course\_chat\_room для присоединения студентов к чат-комнате конкретного курса.

- Используем декоратор @login\_required для защиты доступа неаутентифицированных пользователей.

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

from django.http import HttpResponseForbidden

from django.contrib.auth.decorators import login\_required

@login\_required

def course\_chat\_room(request, course\_id):

try:

course = request.user.courses\_joined.get(id=course\_id)

except:

return HttpResponseForbidden()

return render(request, 'chat/room.html', {'course': course})

2. Настройка URL-адресов. В приложении chat создаем файл urls.py и определяем шаблон URL для course\_chat\_room.

from django.urls import path

from . import views

app\_name = 'chat'

urlpatterns = [

path('room/<int:course\_id>/', views.course\_chat\_room, name='course\_chat\_room'),

]

3. Добавляем шаблоны URL-адресов из приложения chat в основные URL-адреса проекта education для обработки запросов к чат-комнате.

4. Создаем шаблон room.html внутри templates/chat/ с элементами для отображения сообщений и ввода текста сообщения.

{% extends "base.html" %}

{% block title %}Chat room for "{{ course.title }}"{% endblock %}

{% block content %}

<div id="chat">

</div>

<div id="chat-input">

<input id="chat-message-input" type="text">

<input id="chat-message-submit" type="submit" value="Send">

</div>

{% endblock %}

После запуска сервера и перехода по URL чат-комнаты, студенты смогут общаться в чате курса. Таким образом, реализованы представление и шаблон для чат-комнаты веб-чата с помощью Django.

**Установка приложения-обёртки Django Channels**

1. Установи пакет channels в виртуальной среде с помощью **pip install channels**.

2. Настройка в settings.py: добавить channels в INSTALLED\_APPS в файле settings.py.

3. Отредактируй файл asgi.py в проекте, добавив маршрутизацию для работы с HTTP-протоколом через Channels.

4. Добавь в settings.py строку: ASGI\_APPLICATION = 'education.routing.application', где определяется конфигурация маршрутизации корня.

5. Для запуска сервера разработки выполнить: python manage.py runserver.

После успешного запуска сервера Channels, он способен обрабатывать синхронные и асинхронные запросы, включая веб-сокеты.

Теперь можно приступить к разработке чат-сервера используя Channels для:

1. Настройки потребителя для обработки сообщений через WebSocket.

2. Конфигурации маршрутизации для связи URL адресов с потребителями.

3. Создания WebSocket-клиента для обмена сообщениями через JavaScript.

4. Активации канального слоя с использованием Redis для обмена данными между экземплярами приложения.

Таким образом, установлен и сконфигурирован Django Channels для разработки веб-приложения, способного обрабатывать разнообразные типы запросов, включая WebSocket.

**Пишем Потребителя: Интро в Асинхронное Веб-Программирование с Django Channels**

В Django у нас есть Views для работы с HTTP-запросами, и сейчас добавляем Consumers для работы с веб-сокетами. Consumers похожи на Views, но для веб-сокетов.

Давай напишем простого Consumer для эхо сообщений от клиента через веб-сокеты. Создадим файл consumers.py внутри каталога chat:

import json

from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):

def connect(self):

self.accept()

def disconnect(self, close\_code):

pass

def receive(self, text\_data):

text\_data\_json = json.loads(text\_data)

message = text\_data\_json['message']

self.send(text\_data=json.dumps({'message': message}))

Этот ChatConsumer наследуется от WebsocketConsumer из Channels. Важные методы:

- connect(): устанавливает соединение.

- disconnect(): обрабатывает разрыв соединения.

- receive(): обрабатывает полученные данные. Тут мы принимаем сообщение и отправляем обратно через веб-сокет.

Этот Consumer возвращает полученные сообщения. Можно дополнить функционал для широковещательной рассылки сообщений другим клиентам.

**Разбираемся в Маршрутизации для Веб-Сокетов в Django Channels**

Нам нужно, чтобы наши веб-сокеты знали, куда направляться, чтобы они достигли нужного "Потребителя" (Consumer) — это такой Django Channels' термин для обработчика веб-сокетов. Мы для этого используем классы маршрутизации.

Чтобы направить наше соединение к "Потребителю" ChatConsumer, мы создаем файл routing.py внутри папки нашего приложения chat и помещаем туда следующий код:

from django.urls import re\_path

from . import consumers

websocket\_urlpatterns = [

re\_path(r'ws/chat/room/(?P<course\_id>\d+)/$', consumers.ChatConsumer.as\_asgi()),

]

В этом кусочке кода мы связываем URL-адрес с классом ChatConsumer, который мы определили в файле consumers.py. re\_path в Django позволяет нам создать URL-путь с использованием регулярных выражений. В URL-адресе есть цифровой параметр course\_id, который будет передан "Потребителю" для определения комнаты чата пользователя.

Чтобы Django мог различать запросы веб-сокетов от обычных HTTP-запросов, хорошей практикой является добавление префикса /ws/ к URL-адресам веб-сокетов.

Теперь необходимо отредактировать глобальный файл asgi.py, расположенный рядом с settings.py, следующим образом:

import os

from django.core.asgi import get\_asgi\_application

from channels.routing import ProtocolTypeRouter, URLRouter

from channels.auth import AuthMiddlewareStack

import chat.routing

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'educa.settings')

django\_asgi\_app = get\_asgi\_application()

application = ProtocolTypeRouter({

'http': django\_asgi\_app,

'websocket': AuthMiddlewareStack(

URLRouter(chat.routing.websocket\_urlpatterns)

),

})

В этом обновленном коде мы добавляем новый маршрут для протокола веб-сокетов, используя URLRouter для связывания веб-сокет соединений с URL-шаблонами, определенными в списке websocket\_urlpatterns из файла routing.py. Мы также используем AuthMiddlewareStack для стандартной аутентификации Django, обеспечивая доступ к информации о пользователе, что помогает идентифицировать отправителя сообщений в "Потребителе".

**Создание WebSocket Клиента в Django с JavaScript: Моя История**

Нужно создадать WebSocket-клиента, чтобы наш Чат работал как надо.

Итак, я уже создала вьюху course\_chat\_room и связанный с ней шаблон, где ты можешь войти в комнату чата курса. Я также реализовала WebSocket-потребителя для чат-сервера и добавила его в маршрутизации URL-адресов. Теперь пришло время создать WebSocket-клиента, который будет устанавливать связь с веб-сокетом в шаблоне чата и обмениваться сообщениями.

Для этого я использовала JavaScript, чтобы создать и управлять соединением непосредственно в браузере. Да, я общался с DOM через JavaScript.

Посмотрим, как я обновила шаблон room.html внутри приложения chat, немного изменив блоки include\_js и domready:

{% block include\_js %}

{{ course.id|json\_script:"course-id" }}

{% endblock %}

{% block domready %}

const courseId = JSON.parse(document.getElementById('course-id').textContent);

const url = 'ws://' + window.location.host + '/ws/chat/room/' + courseId + '/';

const chatSocket = new WebSocket(url);

{% endblock %}

В блоке include\_js я использовала шаблонный фильтр json\_script, чтобы безопасно работать с course.id в JavaScript. Этот фильтр Django преобразует объект Python в JSON-формат, обернутый тегом script, что позволяет безопасное использование его в JavaScript.

В блоке domready я создала URL для веб-сокета с префиксом ws://, добавив текущий хост браузера из window.location.host и путь к комнате чата, заданный в файле маршрутизации приложения chat.

Теперь просто выполним инструкцию new WebSocket(url), чтобы установить веб-сокет соединение с указанным URL. Полученный объект WebSocket я сохранил в chatSocket.

Итак, у меня есть WebSocket-потребитель, настроенная маршрутизация и базовый WebSocket-клиент.

После того, как я зашла на http://127.0.0.1:8000/chat/room/5/ в браузере, сменила 5 на реальный ID курса из базы данных и наблюдала за консолью. Помимо обычных HTTP запросов и статических файлов, я должна была увидеть две строки, включая WebSocket HANDSHAKING и WebSocket CONNECT.

Так, в сервере разработки Channels используется стандартный TCP-сокет для прослушивания входящих веб-сокет-соединений. Рукопожатие — это своего рода мостик от HTTP к веб-сокетам. Во время рукопожатия согласовываются детали соединения, и любая из сторон может закрыть соединение. Напомню, что в методе connect() класса ChatConsumer, реализованным в файле consumers.py, для принятия любого соединения используется self.accept().

Теперь я готова не только подключаться к веб-сокету, но и начинать взаимодействовать с ним. Давай сделаем наш чат более интерактивным! Вот что я сделала.

Я настроила методы для обработки обычных действий, таких как прием и отправка сообщений, в шаблоне room.html в приложении chat, дополнив блок domready:

{% block domready %}

const courseId = JSON.parse(document.getElementById('course-id').textContent);

const url = 'ws://' + window.location.host + '/ws/chat/room/' + courseId + '/';

const chatSocket = new WebSocket(url);

chatSocket.onmessage = function(event) {

const data = JSON.parse(event.data);

const chat = document.getElementById('chat');

chat.innerHTML += '<div class="message">' + data.message + '</div>';

chat.scrollTop = chat.scrollHeight;

};

chatSocket.onclose = function(event) {

console.error('Chat socket closed unexpectedly');

};

{% endblock %}

В этом коде я определила следующие события для WebSocket-клиента:

- onmessage: вызывается при получении данных через веб-сокет. Я парсил JSON сообщение, извлекал атрибут message и добавлял его как новый элемент <div> в чат. Таким образом, новые сообщения добавлялись к чату, и блок чата автоматически прокручивался к последнему сообщению, делая его видимым.

- onclose: запускается при закрытии соединения с веб-сокетом. В случае неожиданного закрытия выводилось сообщение об ошибке "Chat socket closed unexpectedly".

Теперь я научилась отображать полученные сообщения в чате. Но осталось добавить возможность отправлять сообщения через сокет.

Для этого я добавила следующий JavaScript код в нижнюю часть блока domready в шаблоне chat/room.html приложения chat:

const input = document.getElementById('chat-message-input');

const submitButton = document.getElementById('chat-message-submit');

submitButton.addEventListener('click', function(event) {

const message = input.value;

if (message) {

// отправить сообщение в формате JSON

chatSocket.send(JSON.stringify({'message': message}));

// очистить поле ввода

input.innerHTML = '';

input.focus();

}

});

Теперь при нажатии на кнопку "Submit" сообщение из поля ввода отправляется через веб-сокет. Я конвертировала его в JSON и отправила. После отправки сообщения я очищала поле ввода и возвращала фокус туда для удобства последующего ввода.

Таким образом, я добавила функциональность отправки сообщений через веб-сокет, делая наш чат более интерактивным и захватывающим.

Теперь я могу отправлять сообщения, используя поле ввода текста и нажимая кнопку отправки.

Для улучшения пользовательского опыта я добавила автоматическое установление фокуса в поле ввода текста сразу после загрузки страницы, чтобы можно было сразу начать печатать. Также я отслеживала нажатия клавиш клавиатуры, чтобы при нажатии Enter инициировалось событие click на кнопке отправки. Теперь можно отправлять сообщение, нажимая Enter или кликая по кнопке.

Я добавила следующий JavaScript код в шаблон chat/room.html в приложении chat:

input.addEventListener('keypress', function(event) {

if (event.key === 'Enter') {

event.preventDefault();

submitButton.click();

}

});

input.focus();

Теперь после загрузки страницы фокус автоматически переходит к полю ввода текста. Пользователь сразу видит установленный фокус и может начать вводить сообщение.

Поправим блок domready в шаблоне chat/room.html. Теперь он выглядит так:

{% block domready %}

const courseId = JSON.parse(document.getElementById('course-id').textContent);

const url = 'ws://' + window.location.host + '/ws/chat/room/' + courseId + '/';

const chatSocket = new WebSocket(url);

// Другие события chatSocket...

const input = document.getElementById('chat-message-input');

const submitButton = document.getElementById('chat-message-submit');

submitButton.addEventListener('click', function(event) {

const message = input.value;

if (message) {

chatSocket.send(JSON.stringify({'message': message}));

input.value = '';

input.focus();

}

});

input.addEventListener('keypress', function(event) {

if (event.key === 'Enter') {

event.preventDefault();

submitButton.click();

}

});

input.focus();

{% endblock %}

Теперь зайдём на страницу http://127.0.0.1:8000/chat/room/5/ в своем браузере, введи сообщение и нажми кнопку "SEND" или клавишу "Enter".

Ты увидишь, как твое сообщение появится в чате через веб-сокеты, делая весь процесс более интерактивным и захватывающим.

**Интеграция Канального Слоя для Мультиплексирования Сообщений**

В веб-разработке нередко требуется обеспечить связь между различными частями приложения. И здесь на помощь приходят канальные слои! Они служат транспортным механизмом для обмена сообщениями между различными экземплярами приложения, обеспечивая их взаимодействие.

Каналы и Группы: Организация Сообщений

Каналы выступают как "Входящие" точки, где сообщения могут быть направлены. Каждый канал имеет уникальное имя, и сообщения, отправленные в этот канал, будут получены всеми, кто подписан на него.

Группы, в свою очередь, объединяют несколько каналов. Группе также присваивается имя. Добавляя или удаляя каналы из группы, можно направлять сообщения всем каналам в этой группе.

При построении чат-сервера мы используем группы каналов. Создав группу для каждой чат-комнаты курса, экземпляры ChatConsumer смогут взаимодействовать между собой, обеспечивая передачу сообщений между участниками чата.

Активация Канального Слоя с Redis

С помощью Redis в качестве хранилища, канальный слой работает как почтовая система для обмена сообщениями. Channels предоставляет драйвер для Redis, который отлично подходит для этой задачи.

Установка и Конфигурация Redis для Channels

1. Установим пакет channels-redis в своем виртуальном окружении **pip install channels-redis**.

2. Добавим код в файл settings.py проекта:

CHANNEL\_LAYERS = {

'default': {

'BACKEND': 'channels\_redis.core.RedisChannelLayer',

'CONFIG': {

'hosts': [('127.0.0.1', 6379)],

},

},

}

3. Инициализируем контейнер Redis через Docker **docker run -it --rm --name redis -p 6379:6379 redis**.

4. Запусти Django shell **python manage.py shell**.

5. Проверь работоспособность канального слоя, отправив сообщение в тестовый канал и получив его обратно:

>>> import channels.layers

>>> from asgiref.sync import async\_to\_sync

>>> channel\_layer = channels.layers.get\_channel\_layer()

>>> async\_to\_sync(channel\_layer.send)('test\_channel', {'message': 'hello'})

>>> async\_to\_sync(channel\_layer.receive)('test\_channel')

Теперь мы готовы отправлять и принимать сообщения через канальный слой, обеспечивая удобное взаимодействие между различными частями приложения.

**Широковещательная Рассылка Сообщений**

Мы обновляем потребителя для широковещательной рассылки сообщений. Давай настраивать его, чтобы быть в курсе всего, что происходит в чате.

Открываем consumers.py в приложении chat, добавляем async\_to\_sync() и редактируем метод connect() в ChatConsumer:

from channels.generic.websocket import WebsocketConsumer

from asgiref.sync import async\_to\_sync

class ChatConsumer(WebsocketConsumer):

def connect(self):

self.id = self.scope['url\_route']['kwargs']['course\_id']

self.room\_group\_name = f'chat\_{self.id}'

# Присоединяемся к группе чата

async\_to\_sync(self.channel\_layer.group\_add)(

self.room\_group\_name,

self.channel\_name

)

# Принимаем соединение

self.accept()

Обновления в методе connect():

- Разбираем id курса для определения связи с чат-комнатой.

- Формируем имя группы для каждой учебной комнаты.

- Присоединяемся к группе, добавляя текущий канал.

- self.accept() принимает WebSocket-соединение.

Метод disconnect():

class ChatConsumer(WebsocketConsumer):

# ...

def disconnect(self, close\_code):

# Покидаем группу чата

async\_to\_sync(self.channel\_layer.group\_discard)(

self.room\_group\_name,

self.channel\_name

)

# ...

- При отключении, покидаем группу чата через group\_discard().

- Обновления происходят с использованием async\_to\_sync() для асинхронных методов.

Давай рассмотрим обновления в методе receive() класса ChatConsumer в файле consumers.py.

class ChatConsumer(WebsocketConsumer):

# ...

def receive(self, text\_data):

text\_data\_json = json.loads(text\_data)

message = text\_data\_json['message']

async\_to\_sync(self.channel\_layer.group\_send)(

self.room\_group\_name,

{'type': 'chat\_message',

'message': message, } )

- Получаем сообщение из WebSocket.

- Отправляем сообщение всей группе, а не только определенному каналу, используя group\_send() из канального слоя.

Дополнительный метод chat\_message():

class ChatConsumer(WebsocketConsumer):

# ...

def chat\_message(self, event):

self.send(text\_data=json.dumps(event))

- Метод chat\_message() соответствует ключу type, отправленному в канальную группу при получении сообщения из WebSocket.

- При отправке сообщения с типом chat\_message в группу, все подписанные потребители выполняют этот метод и получают сообщение.

Теперь наш ChatConsumer имеет возможности широковещательной рассылки сообщений и обмена мнениями.

Перейдём в своем браузере по ссылке http://127.0.0.1:8000/chat/room/5/, где цифру 5 это ID курса из базы данных. Пишем и отправляем сообщения, наблюдая, как они летят во вселенную.

В consumers.py мы внесли изменения для отображения данных пользователей и времени в чате:

import json

from channels.generic.websocket import WebsocketConsumer

from asgiref.sync import async\_to\_sync

from django.utils import timezone

class ChatConsumer(WebsocketConsumer):

def connect(self):

self.user = self.scope['user']

self.id = self.scope['url\_route']['kwargs']['course\_id']

self.room\_group\_name = f'chat\_{self.id}'

async\_to\_sync(self.channel\_layer.group\_add)(self.room\_group\_name, self.channel\_name)

self.accept()

def disconnect(self, close\_code):

async\_to\_sync(self.channel\_layer.group\_discard)(self.room\_group\_name, self.channel\_name)

def receive(self, text\_data):

text\_data\_json = json.loads(text\_data)

message = text\_data\_json['message']

now = timezone.now()

async\_to\_sync(self.channel\_layer.group\_send)(

self.room\_group\_name,

{

'type': 'chat\_message',

'message': message,

'user': self.user.username,

'datetime': now.isoformat(),

}

)

def chat\_message(self, event):

self.send(text\_data=json.dumps(event))

В методе connect() мы узнаем текущего пользователя и используем модуль timezone для текущего времени. Метод receive() передает сообщение, пользователя и время в группу через channel\_layer.

В шаблоне room.html в блоке domready добавим изменения:

{% block include\_js %}

{{ course.id|json\_script:"course-id" }}

{{ request.user.username|json\_script:"request-user" }}

{% endblock %}

{% block domready %}

const courseId = JSON.parse(document.getElementById('course-id').textContent);

const requestUser = JSON.parse(document.getElementById('request-user').textContent);

// ...

{% endblock %}

Теперь мы безопасно извлекаем данные пользователя и сохраняем их в requestUser. Это добавляет личность и время к каждому сообщению в чате, что делает общение интересным и удобным.

Внесем изменения в блок domready:

const data = JSON.parse(e.data);

const chat = document.getElementById('chat');

const dateOptions = {hour: 'numeric', minute: 'numeric', hour12: true};

const datetime = new Date(data.datetime).toLocaleString('en', dateOptions);

const isMe = data.user === requestUser;

const source = isMe ? 'me' : 'other';

const name = isMe ? 'Me' : data.user;

chat.innerHTML += '<div class="message ' + source + '">' +

'<strong>' + name + '</strong> ' +

'<span class="date">' + datetime + '</span><br>' +

data.message + '</div>';

chat.scrollTop = chat.scrollHeight;

В этом обновленном коде мы добавляем информацию о пользователе и времени к каждому сообщению, делая чат более информативным и интересным

Мы выполняем следующие шаги на JavaScript:

- Преобразовываем полученную дату и время в объект JavaScript Date и форматируем его с параметрами локали.

- Проверяем, отправил ли сообщение текущий пользователь или кто-то другой.

- Определяем класс сообщения (me или other) на основе отправителя.

- Формируем имя отправителя, чтобы отобразить Me или имя пользователя.

- Применяем класс к основному элементу `<div>`, содержащему сообщение, чтобы стилизовать его в соответствии с отправителем. Стили заданы в файле `css/base.css`.

- Добавляем к сообщению имя пользователя и дату и время, чтобы сообщения были красиво отформатированы и отличались в зависимости от отправителя.

После внесения этих изменений в JavaScript, предлагаю открыть страницу чата по адресу http://127.0.0.1:8000/chat/room/5/. Затем отправьте сообщения от двух разных пользователей — одного обычного и другого в режиме инкогнито.

**Видоизменение потребителя с для обеспечения полной асинхронности**

Раньше мой ChatConsumer был связан с WebsocketConsumer, что было удобно для Django и синхронных функций. Но асинхронные штуки работают просто супер, не надо ломать голову с потоками! Так что, пока мы используем асинхронные функции, давай быстро превратим наш ChatConsumer в крутого асинхронщика.

В файле consumers.py мы вносим правки:

import json

from channels.generic.websocket import AsyncWebsocketConsumer

from django.utils import timezone

class ChatConsumer(AsyncWebsocketConsumer):

async def connect(self):

self.user = self.scope['user']

self.id = self.scope['url\_route']['kwargs']['course\_id']

self.room\_group\_name = 'chat\_%s' % self.id

await self.channel\_layer.group\_add(self.room\_group\_name, self.channel\_name)

await self.accept()

async def disconnect(self, close\_code):

await self.channel\_layer.group\_discard(self.room\_group\_name, self.channel\_name)

async def receive(self, text\_data):

text\_data\_json = json.loads(text\_data)

message = text\_data\_json['message']

now = timezone.now()

await self.channel\_layer.group\_send(

self.room\_group\_name,

{

'type': 'chat\_message',

'message': message,

'user': self.user.username,

'datetime': now.isoformat(),

}

)

async def chat\_message(self, event):

await self.send(text\_data=json.dumps(event))

Вот что изменилось:

- ChatConsumer теперь работает на AsyncWebsocketConsumer для асинхронного магии.

- Все методы теперь объявлены с async def.

- Теперь для работы с асинхронными функциями используется await.

- Теперь можно обойтись без async\_to\_sync() при работе с канальным слоем.

Заходим на страницу чата по адресу http://127.0.0.1:8000/chat/room/5/ в двух вкладках и убедимся, что чат продолжает работать без проблем.

**Интеграция приложения для ведения чата**

Нам нужно добавить немного магии в наши страницы, чтобы студенты с легкостью могли заскочить в чат и делиться мыслями.

В файле detail.html в папке students я добавил волшебную ссылку на чат-комнату курса:

<div class="contents">

<!-- Тут было всякое раньше -->

<h3>

<a href="{% url "chat:course\_chat\_room" object.id %}">

Course chat room

</a>

</h3>

</div>

Теперь когда студент заходит на страницу курса, он увидит ссылку "Course chat room" на боковой панели, которая мгновенно перенесет его в чат-комнату этого курса. Просто пара кликов, и вот - студент уже в мире общения и обмена идеями в реальном времени.

**Дальнейшее развитие проекта**

После успешной разработки платформы для онлайн обучения на фреймворке Django, важным этапом является обеспечение безопасности проекта и его дальнейшее развертывание.

Обеспечение защиты:

1. Обновление зависимостей: Регулярное обновление библиотек и фреймворков поможет избежать уязвимостей, а также повысит стабильность проекта.

2. Защита от атак: Использование механизмов защиты, таких как защита от CSRF, SQL инъекций, XSS атак, поможет предотвратить потенциальные угрозы безопасности.

3. Аутентификация и авторизация: Установка надежной системы аутентификации и авторизации поможет контролировать доступ к ресурсам и защитить конфиденциальные данные пользователей.

4. Мониторинг безопасности: Внедрение системы мониторинга безопасности поможет оперативно обнаруживать и реагировать на возможные инциденты.

Развёртывание проекта:

1. Выбор хостинга: Выбрать надежного провайдера хостинга, который поддерживает деплоймент Django приложений.

2. Автоматизация деплоя: Настроить процесс автоматического деплоя с использованием инструментов типа Docker, Git и Continuous Integration/Delivery.

3. Масштабируемость: Обеспечить возможность горизонтального масштабирования для роста числа пользователей и надёжности системы.

4. Резервное копирование: Регулярно создавать резервные копии данных для предотвращения потери информации в случае сбоев или атак.

Обеспечение безопасности и грамотное развертывание проекта имеет ключевое значение для его успешного функционирования и дальнейшего развития. Правильно настроенная защита и устойчивое развертывание позволят создать надёжную и эффективную платформу для онлайн обучения на основе фреймворка Django.

**Заключение**

В ходе выполнения дипломной работы по теме "Разработка платформы для онлайн обучения на фреймворке Django" была создана функциональная и удобная платформа, предоставляющая возможности для эффективной дистанционной образовательной деятельности. В данной главе будет приведено краткое подведение итогов выполненной работы.

Результаты и достижения

1. Разработка функционала: В результате работы был реализован широкий спектр функциональных возможностей, таких как управление курсами, модулями, заданиями, чатом для общения, система аутентификации и авторизации пользователей.

2. Интерфейс и пользовательский опыт: Был сделан упор на создание удобного и интуитивно понятного интерфейса, который способствует комфортному взаимодействию пользователей с платформой.

3. Технологический стек: Использование фреймворка Django позволило создать стабильное и расширяемое программное обеспечение, обеспечивающее высокую производительность приложения.

Практическая значимость

Разработанная платформа для онлайн обучения на фреймворке Django открывает новые возможности для организации дистанционного образования, позволяя эффективно передавать знания и обеспечивать интегрированное взаимодействие между преподавателями и студентами.

Перспективы развития

Дальнейшее развитие проекта может включать в себя расширение функционала платформы, внедрение новых образовательных инструментов, оптимизацию производительности и безопасности, а также интеграцию с современными технологиями для улучшения пользовательского опыта.

Таким образом, разработка платформы для онлайн обучения на фреймворке Django является значимым шагом в сфере образования, обеспечивая новые возможности для обучения и взаимодействия на основе современных технологий.

**Использованная литература**

1. Использование фикстур Django для тестирования: [Документация Django](https://docs.djangoproject.com/en/5.0/topics/testing/tools/" \l "fixture-loading). Этот источник расскажет тебе о том, как использовать фикстуры в Django для создания примеров данных, которые могут быть использованы в тестах.
2. Миграция данных: [Документация Django](https://docs.djangoproject.com/en/5.0/topics/migrations/" \l "data-migrations).
3. Создание конкретно-прикладных модельных полей: [Документация Django](https://docs.djangoproject.com/en/5.0/howto/custom-model-fields/).
4. [Документация по использованию примесей в классах Django](https://docs.djangoproject.com/en/5.0/topics/class-based-views/mixins/).
5. [Руководство по созданию пользовательских разрешений](https://docs.djangoproject.com/en/5.0/topics/auth/customizing/" \l "custom-permissions).
6. [Django Formsets](https://docs.djangoproject.com/en/5.0/topics/forms/formsets/).
7. [Модельные Formsets в Django](https://docs.djangoproject.com/en/5.0/topics/forms/modelforms/" \l "model-formsets).
8. HTML5 API для перетаскивания элементов: <https://www.w3schools.com/html/html5_draganddrop.asp>.
9. Документация по библиотеке HTML5 Sortable: <https://github.com/lukasoppermann/html5sortable>.
10. Примеры использования библиотеки HTML5 Sortable: <https://lukasoppermann.github.io/html5sortable/>.
11. [Документация по django-braces](https://django-braces.readthedocs.io/).
12. [Документация по django-embed-video](https://django-embed-video.readthedocs.io/en/last/).
13. [Документация по кеш-фреймворку Django](https://docs.djangoproject.com/en/5.0/topics/cache/).
14. [Файлы кеш-бэкенда Memcached для загрузки](https://memcached.org/downloads).
15. [Официальный сайт кеш-бэкенда Memcached](https://memcached.org/).
16. [Исходный код библиотеки pymemcache](https://github.com/pinterest/pymemcache).
17. [Исходный код приложения Django Redisboard](https://github.com/ionelmc/django-redisboard).
18. Официальный сайт фреймворка [Django REST framework](https://www.django-rest-framework.org/).
19. [Настройки Django REST framework](https://www.django-rest-framework.org/api-guide/settings/).
20. [Рендереры Django REST framework](https://www.django-rest-framework.org/api-guide/renderers/).
21. [Парсеры Django REST framework](https://www.django-rest-framework.org/api-guide/parsers/).
22. [Примесные классы и представления Django REST framework](https://www.django-rest-framework.org/api-guide/generic-views/).
23. [Скачать инструмент curl](https://curl.se/download.html).
24. API-платформа Postman: [getpostman.com](https://www.getpostman.com/).
25. Сериализаторы Django REST framework: [django-rest-framework.org/api-guide/serializers](https://www.django-rest-framework.org/api-guide/serializers/).
26. Базовая HTTP-аутентификация: [Wikipedia - Basic access authentication](https://en.wikipedia.org/wiki/Basic_access_authentication).
27. Аутентификация в Django REST framework: [django-rest-framework.org/api-guide/authentication](https://www.django-rest-framework.org/api-guide/authentication/).
28. Разрешения в Django REST framework: [django-rest-framework.org/api-guide/permissions](https://www.django-rest-framework.org/api-guide/permissions/).
29. Наборы представлений ViewSet в Django REST framework: [django-rest-framework.org/api-guide/viewsets](https://www.django-rest-framework.org/api-guide/viewsets/).
30. Маршрутизаторы Django REST framework: [django-rest-framework.org/api-guide/routers](https://www.django-rest-framework.org/api-guide/routers/).
31. Документация по библиотеке Python requests: [requests.readthedocs.io/en/master](https://requests.readthedocs.io/en/master/).
32. Аутентификация с использованием библиотеки requests: [requests.readthedocs.io/en/master/user/authentication](https://requests.readthedocs.io/en/master/user/authentication/).
33. Руководство по ASGI: <https://asgi.readthedocs.io/en/latest/introduction.html>.
34. Поддержка асинхронных представлений в Django: <https://docs.djangoproject.com/en/5.0/topics/async/>.
35. Асинхронные представления на основе классов в Django: [https://docs.djangoproject.com/en/5.0/topics/class-based-views/#async-class-based-views](https://docs.djangoproject.com/en/5.0/topics/class-based-views/" \l "async-class-based-views).
36. Документация по Django Channels: <https://channels.readthedocs.io/>.
37. Развертывание Django с использованием ASGI: <https://docs.djangoproject.com/en/5.0/how-to/deployment/asgi/>.
38. Использование шаблонного фильтра json\_script: [https://docs.djangoproject.com/en/5.0/ref/templates/builtins/#json-script](https://docs.djangoproject.com/en/5.0/ref/templates/builtins/" \l "json-script).
39. Ссылка на код проекта [GitHub](https://github.com/Odarochka1802/DiplomEducation) (https://github.com/Odarochka1802/DiplomEducation).

**Приложения**

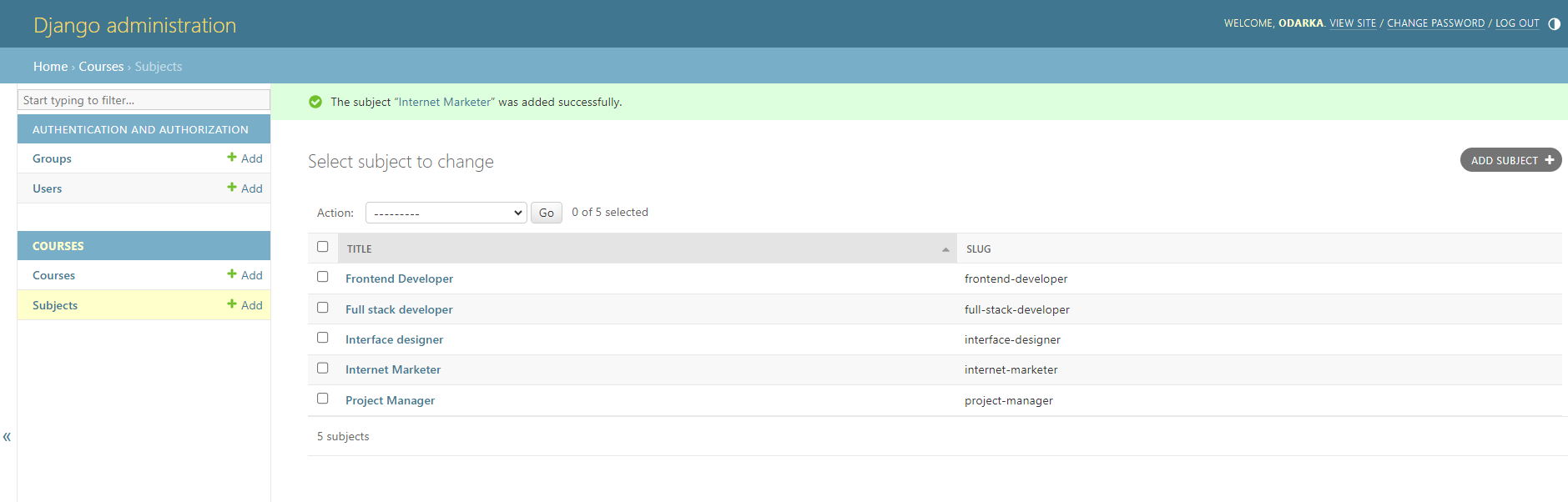


Рис 1

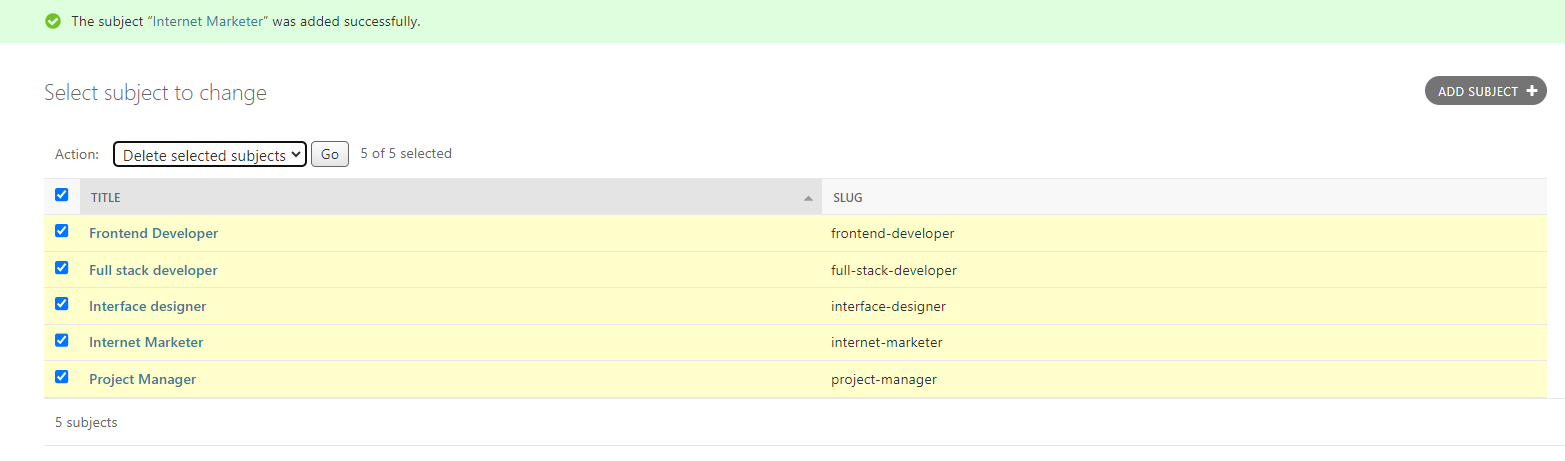


Рис.2

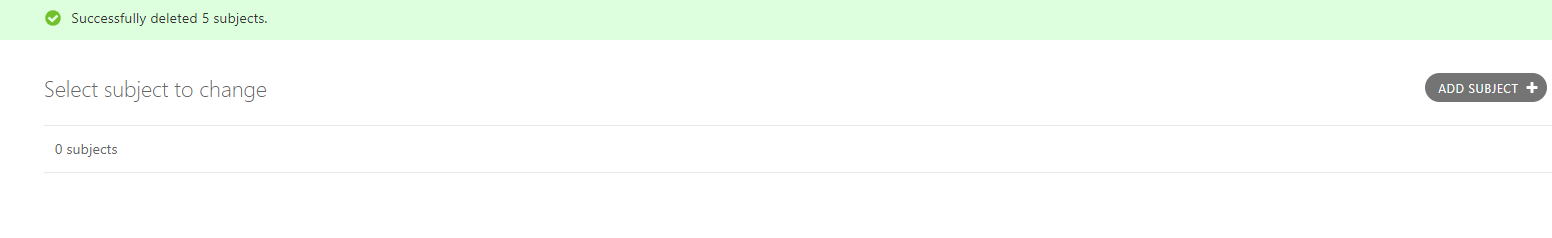


Рис.3

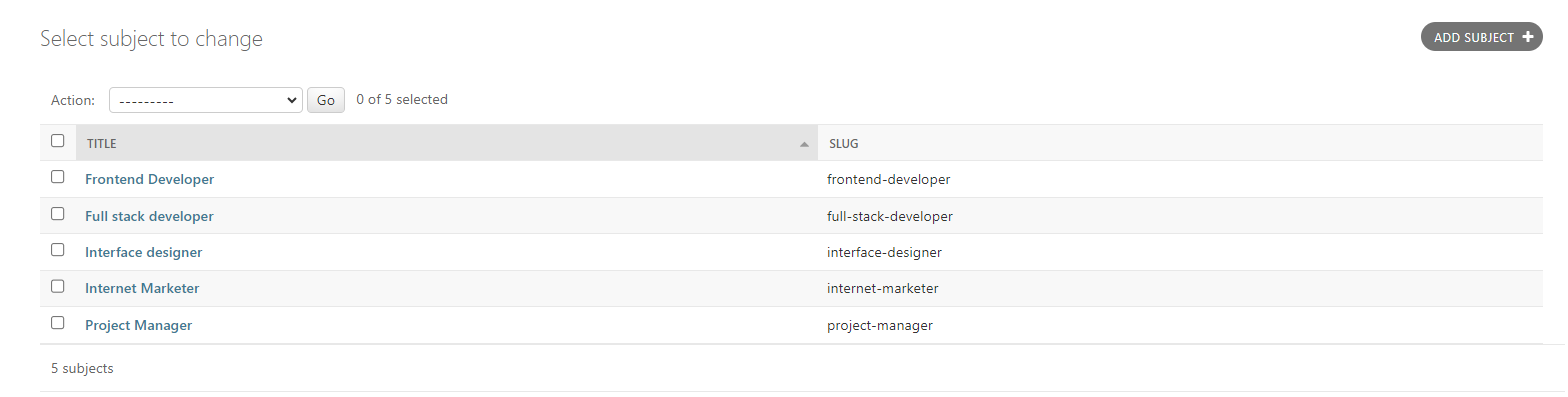
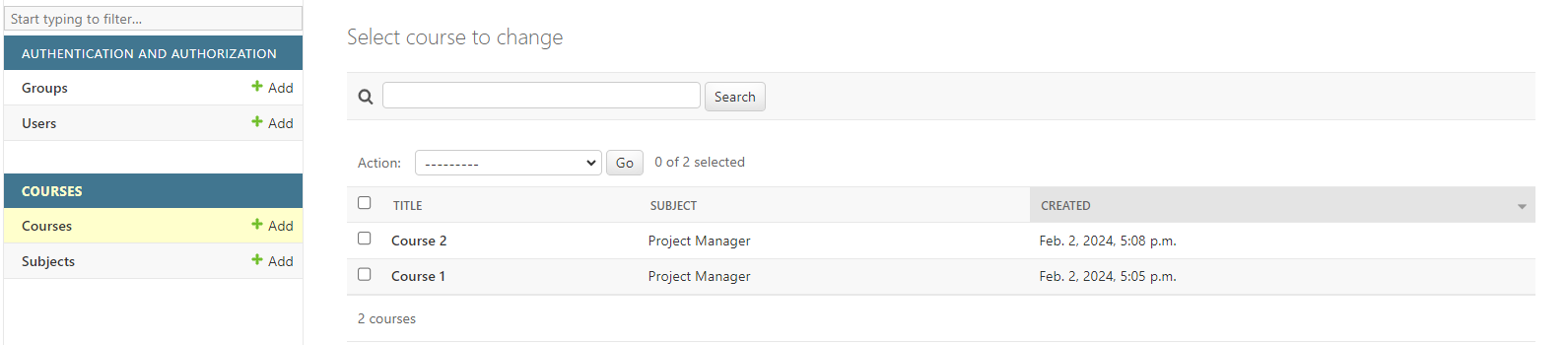


Рис.4

Рис. 5

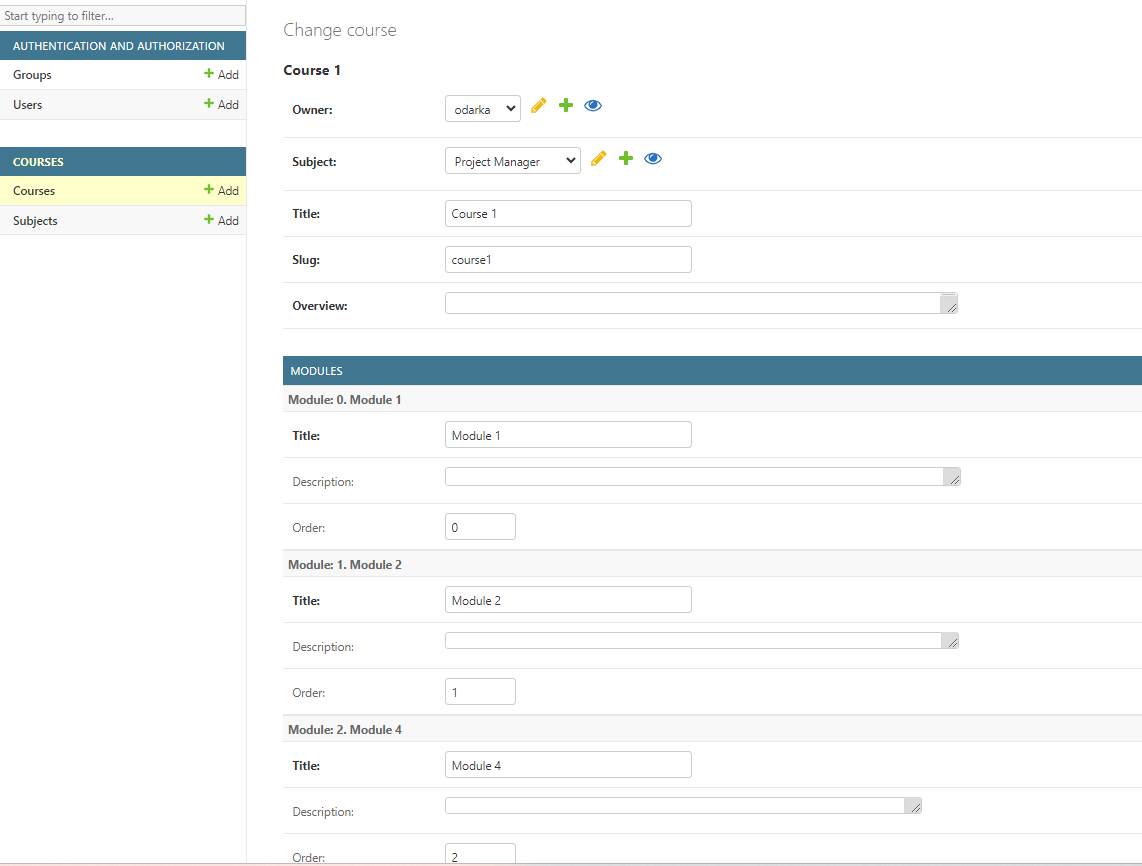


Рис.6

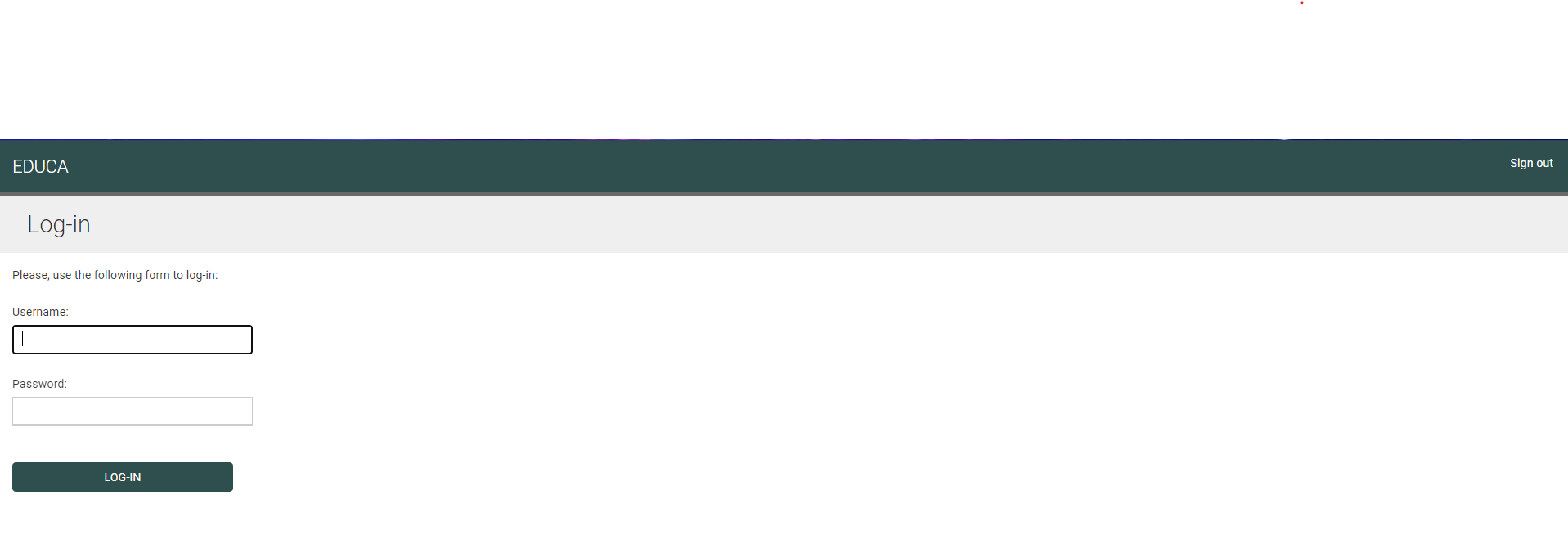


Рис.7

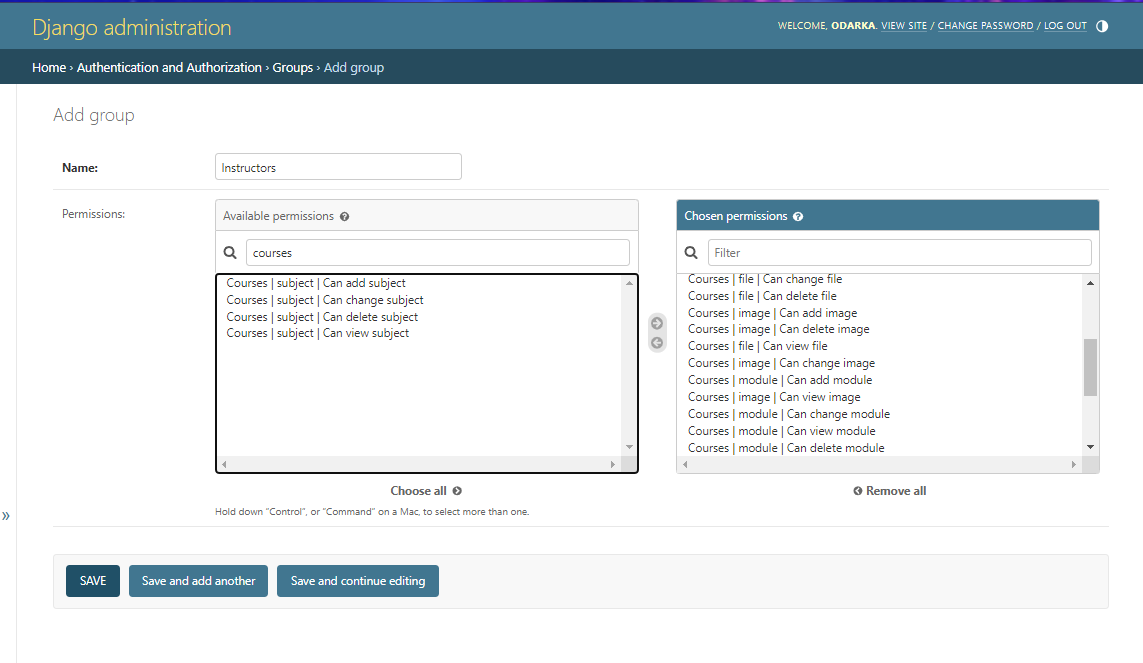
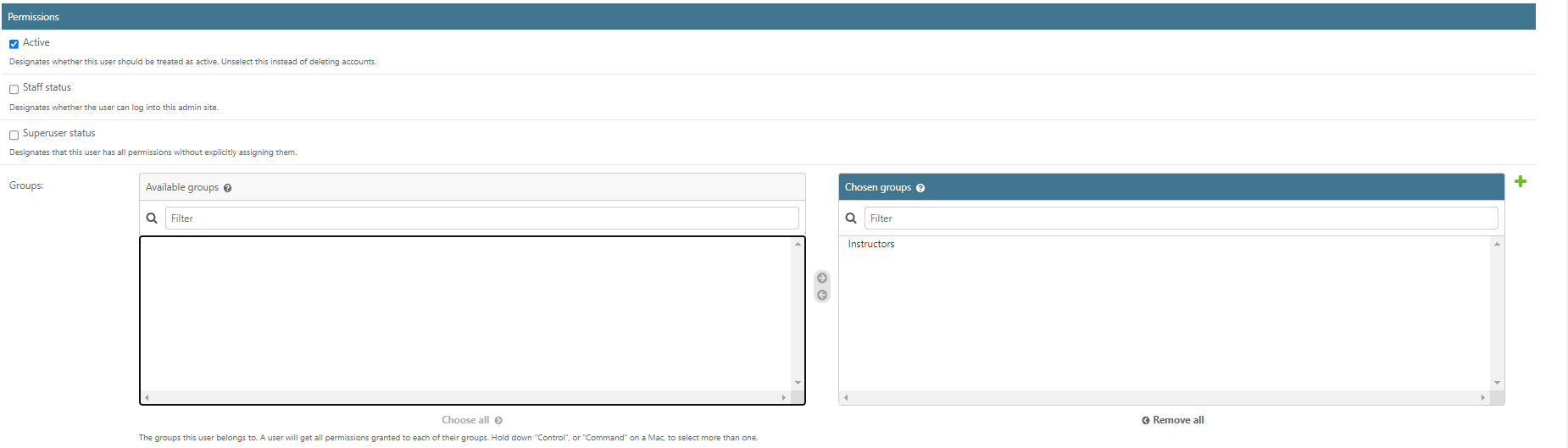


Рис. 8



**Рис.9**

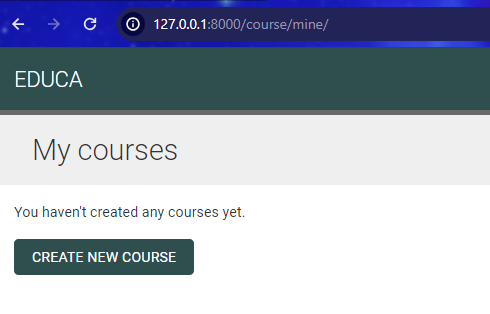


Рис.10

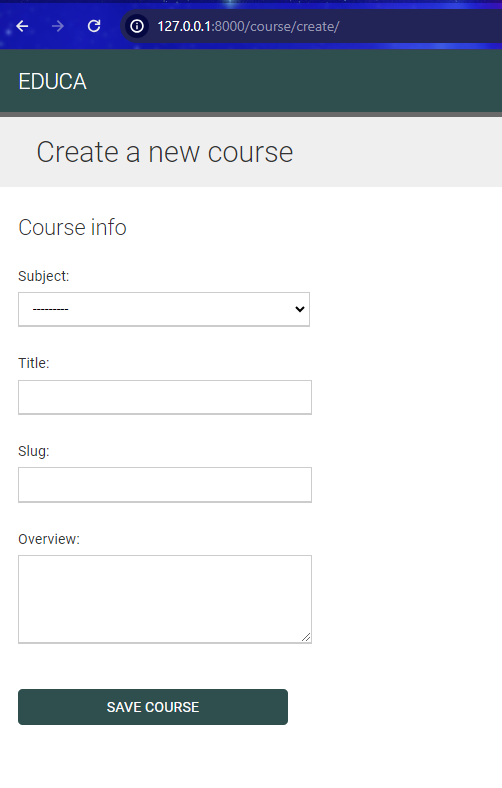


Рис.11

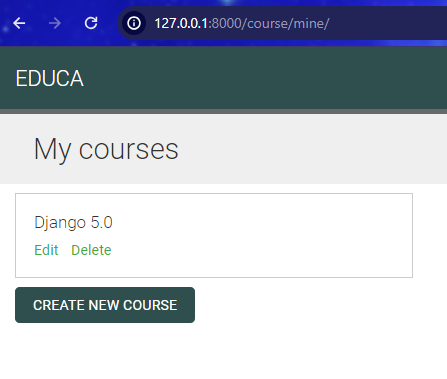
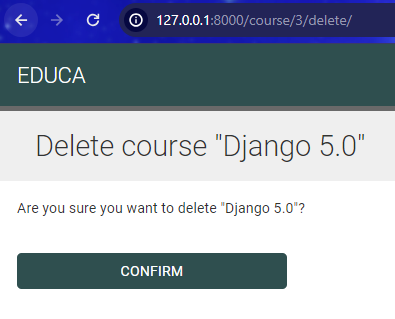
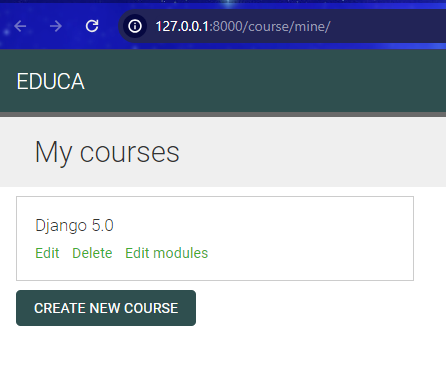


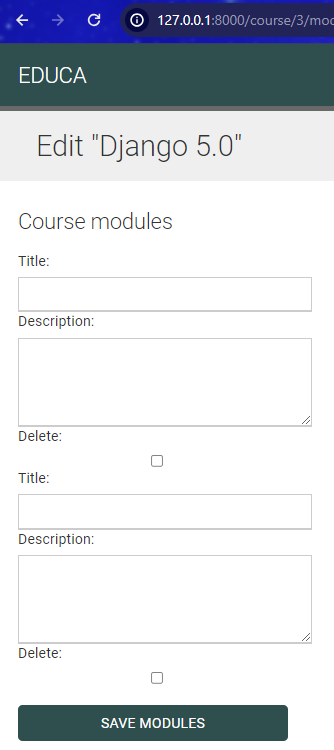
Рис.12



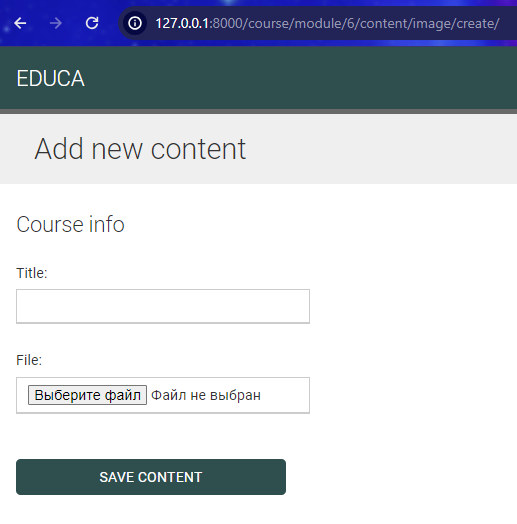
Рыс.13



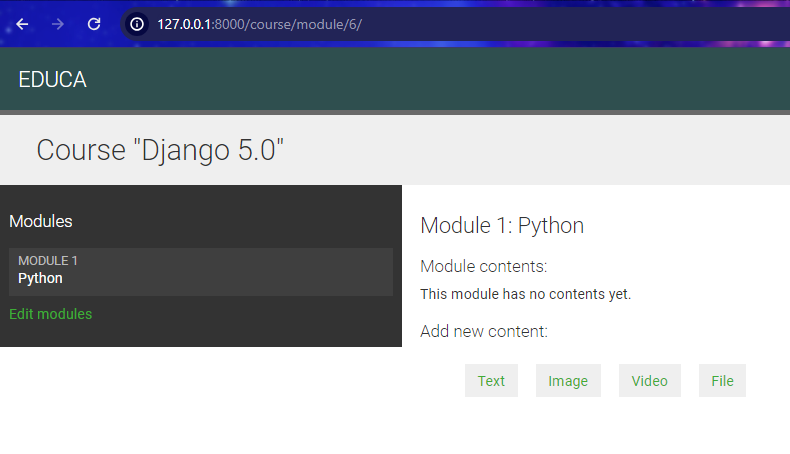
Рыс.14



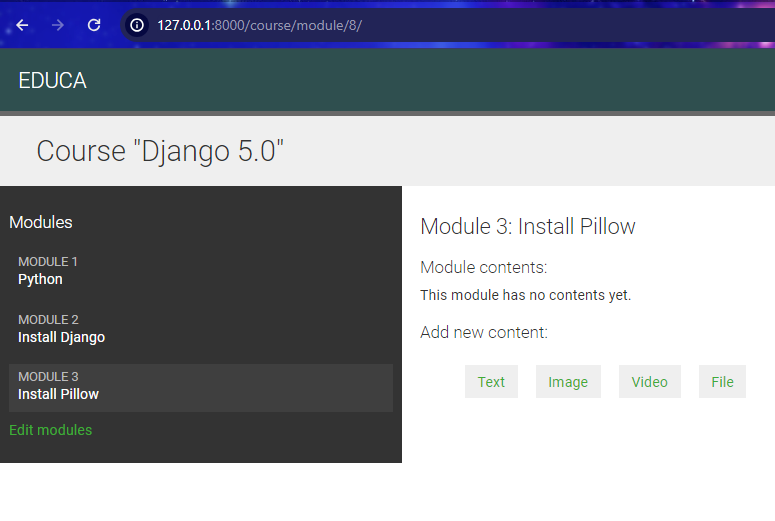
Рыс.15



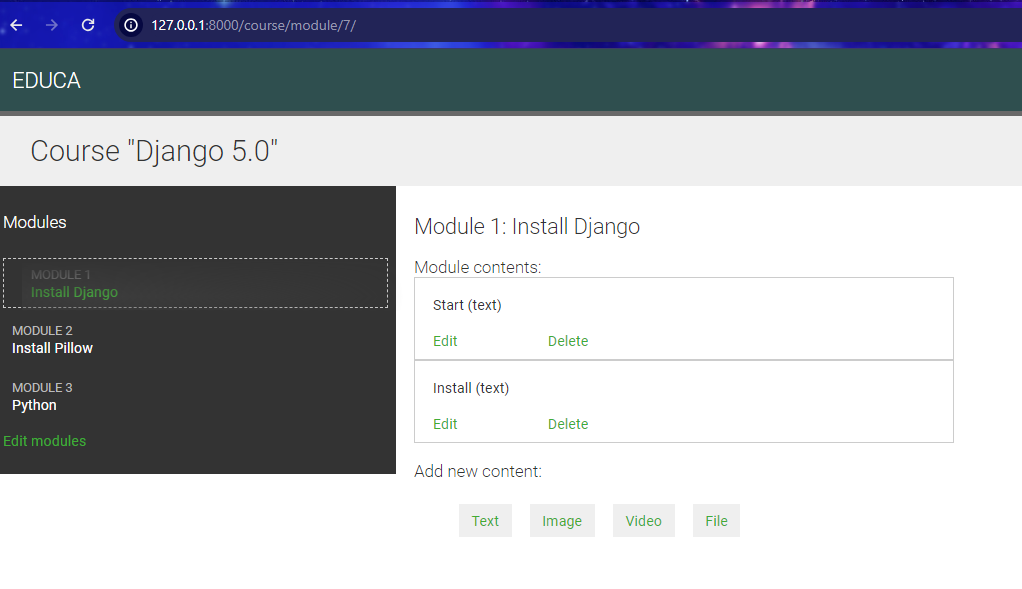
Рыс.16



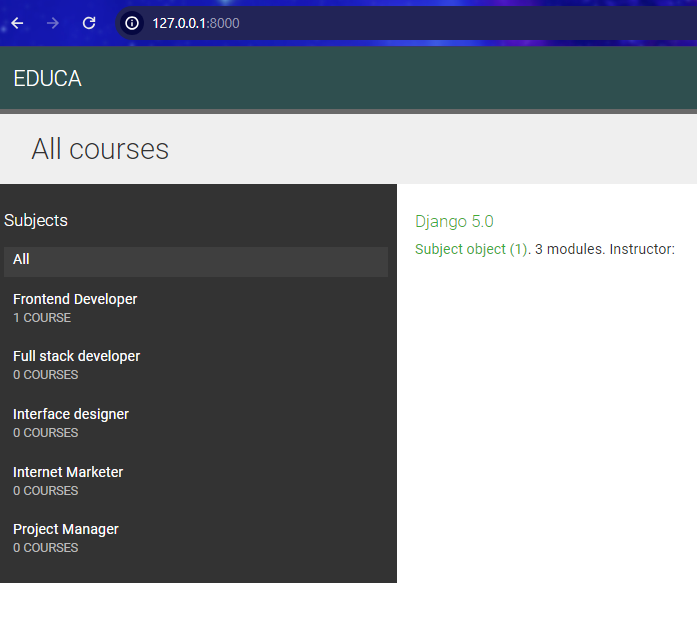
Рыс.17



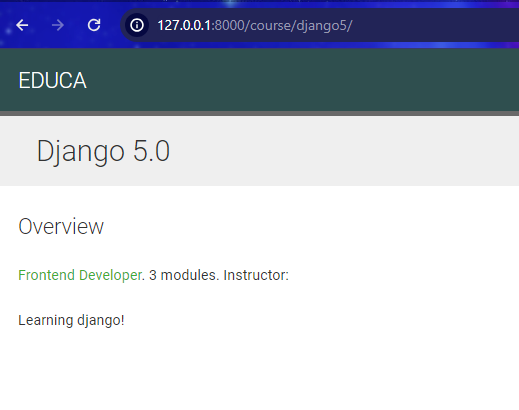
Рыс.18



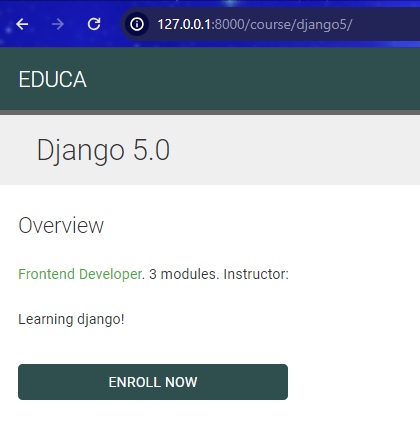
Рыс.19



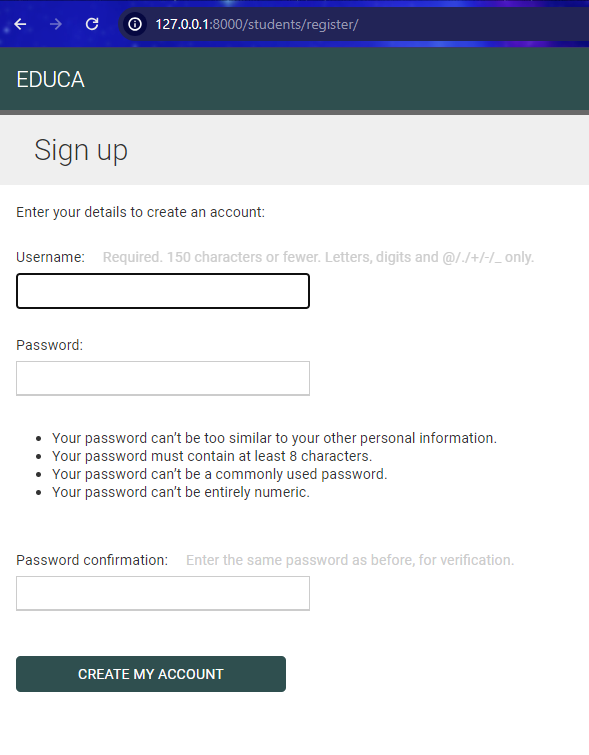
Рыс.20



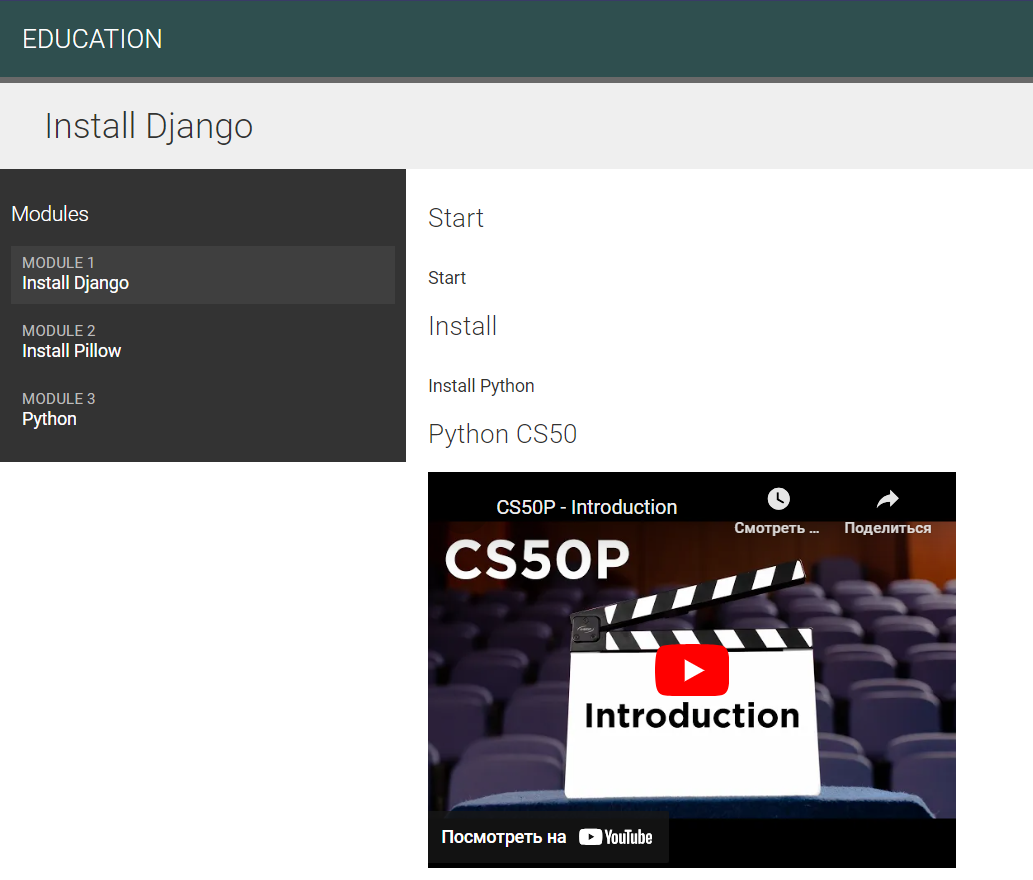
Рыс.21



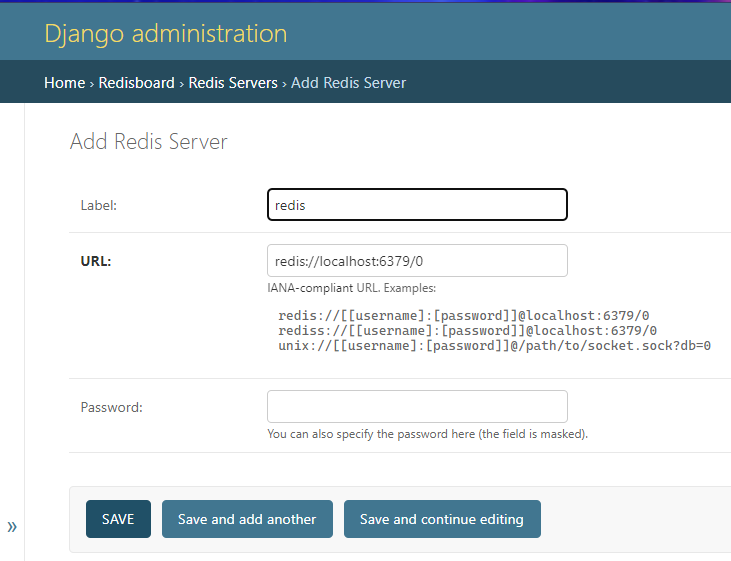
Рыс.22



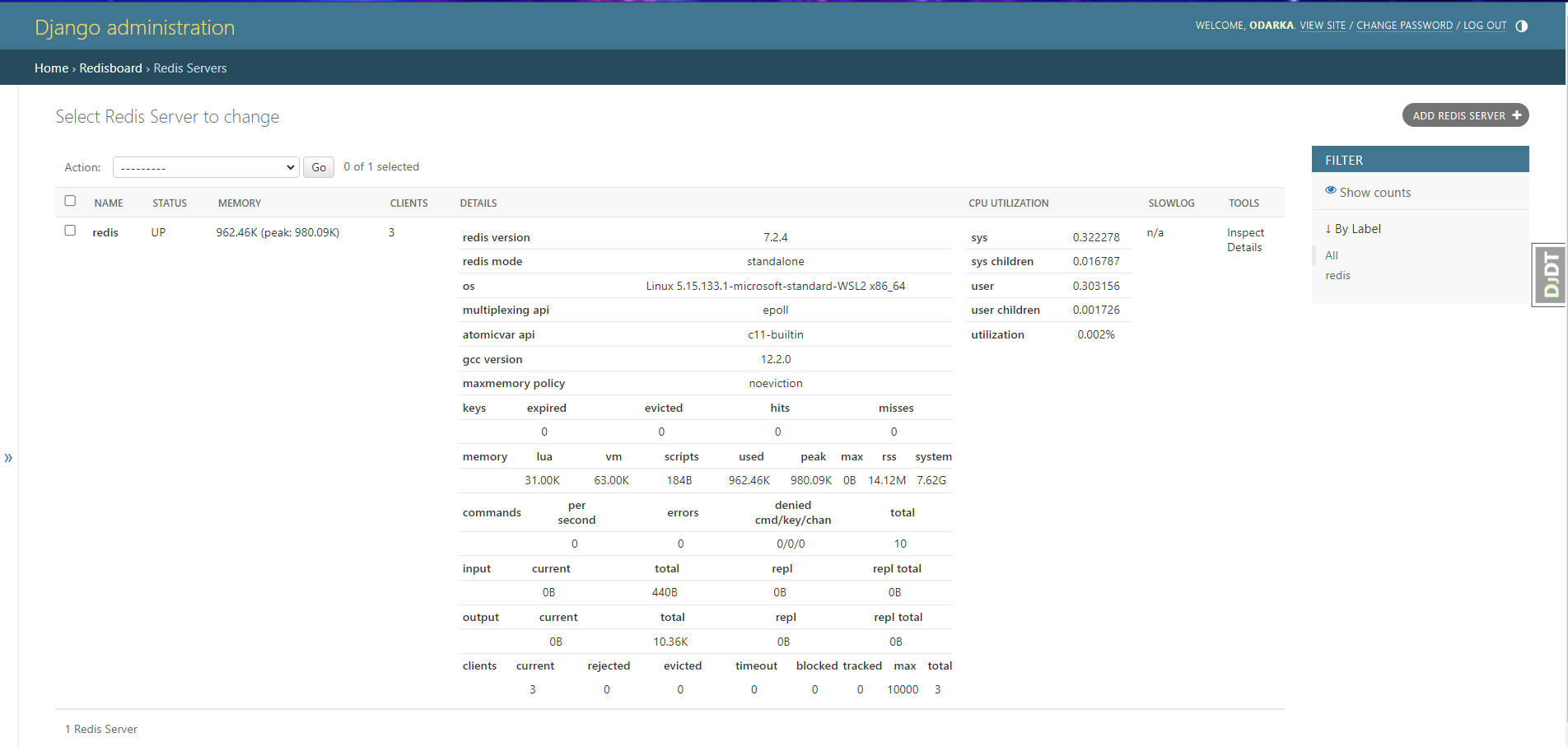
Рыс.23



Рыс. 24



Рыс. 25



Рыс. 26