Building an E-Learning Platform

In this Internship project, you will create a new project. You will build an e-learning platform, creating a custom **Content Management System** (**CMS**).

In this chapter, you will learn how to:

- Create fixtures for your models
- Use model inheritance
- Create custom model fields
- Use class-based views and mixins
- Build formsets
- Manage groups and permissions
- Create a CMS

Setting up the e-learning project

Our final practical project will be an e-learning platform. In this chapter, we are going to build a flexible CMS that allows instructors to create courses and manage their contents.

First, create a virtual environment for your new project and activate it with the following commands:

```
mkdir env
virtualenv env/educa
source env/educa/bin/activate
```

Install Django in your virtual environment with the following command:

```
pip install Django==2.0.5
```

We are going to manage image uploads in our project, so we also need to install `Pillow` with the following command:

```
pip install Pillow==5.1.0
```

Create a new project using the following command:

```
django-admin startproject educa
```

Enter the new `educa` directory and create a new application using the following commands:

```
cd educa
django-admin startapp courses
```

Edit the `settings.py` file of the `educa` project and add `courses` to the `INSTALLED_APPS` setting as follows:

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

The `courses` application is now active for the project. Let's define the models for courses and course contents.

Building the course models

Our e-learning platform will offer courses on various subjects. Each course will be divided into a configurable number of modules, and each module will contain a configurable number of contents. There will be contents of various types: text, file, image, or video. The following example shows what the data structure of our course catalog will look like:

```
Subject 1
  Course 1
    Module 1
      Content 1 (image)
      Content 2 (text)
    Module 2
      Content 3 (text)
      Content 4 (file)
      Content 5 (video)
      ...
```

Let's build the course models. Edit the `models.py` file of the `courses` application and add the following code to it:

```
from django.db import models
from django.contrib.auth.models import User

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
```

These are the initial `Subject`, `Course`, and `Module` models. The `Course` model fields are as follows:

- `owner`: The instructor that created this course.
- `subject`: The subject that this course belongs to. A `ForeignKey` field that points to the `Subject` model.
- `title`: The title of the course.
- `slug`: The slug of the course. This will be used in URLs later.
- `overview`: This is a `TextField` column to include an overview of the course.
- `created`: The date and time when the course was created. It

will be automatically set by Django when creating new objects because of `auto_now_add=True`.

Each course is divided into several modules. Therefore, the `Module` model contains a `ForeignKey` field that points to the `Course` model.

Open the shell and run the following command to create the initial migration for this app:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'courses':
  0001_initial.py:
    - Create model Course
    - Create model Module
    - Create model Subject
    - Add field subject to course
```

Then, run the following command to apply all migrations to the database:

```
python manage.py migrate
```

You should see output including all applied migrations, including those of Django. The output will contain the following line:

```
Applying courses.0001_initial... OK
```

The models of our `courses` app have been synced to the database.

Registering the models in the administration site

Let's add the course models to the administration site. Edit the `admin.py` file inside the `courses` application directory and add the following code to it:

```
from django.contrib import admin
from .models import Subject, Course, Module

@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]
```

The models for the course application are now registered in the administration site. Remember, we use the `@admin.register()` decorator to register models in the administration site.

Using fixtures to provide initial data for models

Sometimes you might want to pre-populate your database with hardcoded data. This is useful to automatically include initial data in the project setup instead of having to add it manually. Django comes with a simple way to load and dump data from the database into files that are called **fixtures**.

Django supports fixtures in JSON, XML, or YAML formats. We are going to create a fixture to include several initial `Subject` objects for our project.

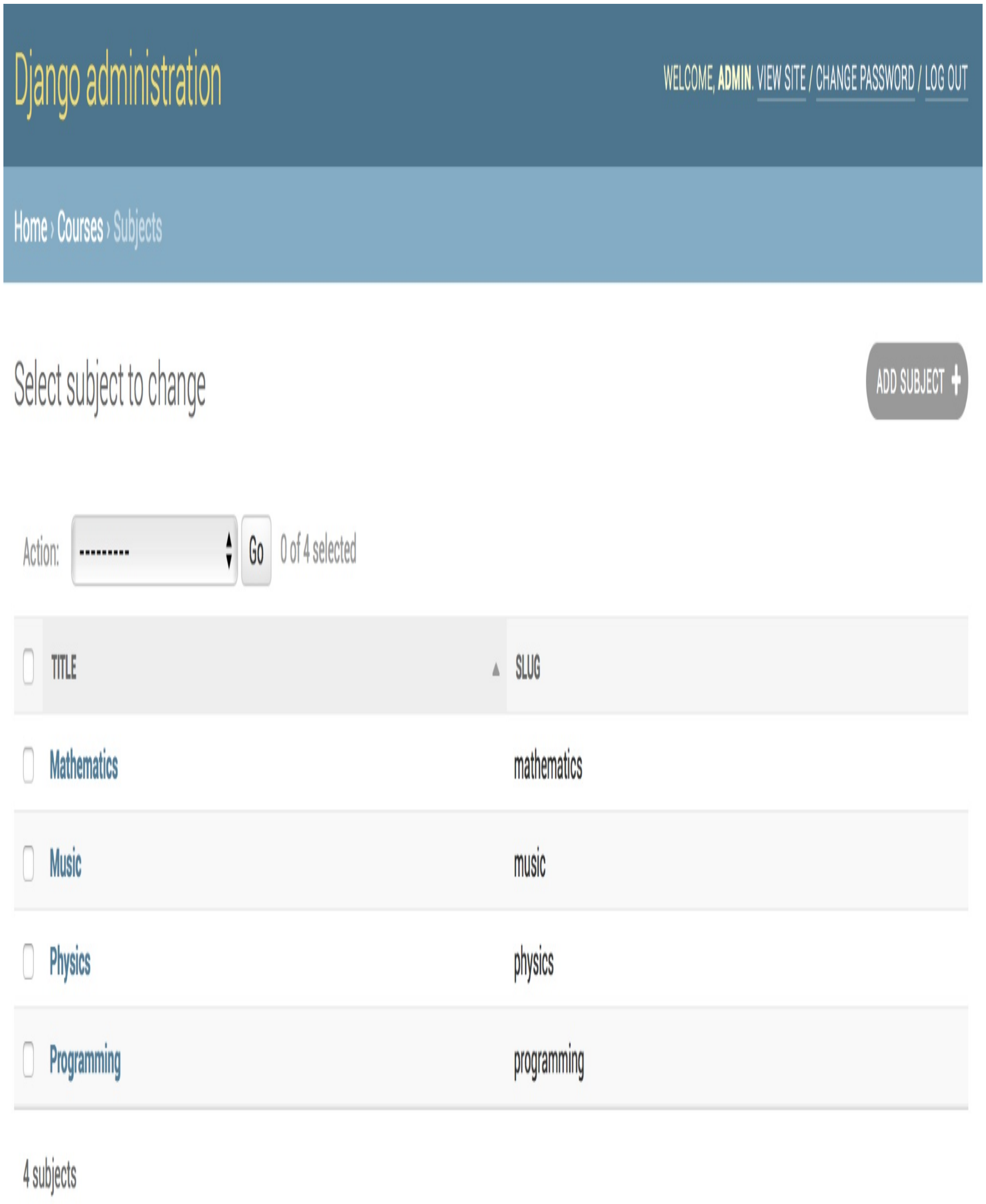
First, create a superuser using the following command:

```
python manage.py createsuperuser
```

Then, run the development server using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/courses/subject/` in your browser. Create several subjects using the administration site. The list display page should look as follows:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Courses › Subjects
ADD SUBJECT
Action: ---------
Go

Select subject to change

0 of 4 selected

TITLE	SLUG
Mathematics	mathematics
Music	music
Physics	physics
Programming	programming

4 subjects

Run the following command from the shell:

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

You will see output similar to the following:

```
[
{
  "model": "courses.subject",
  "pk": 1,
  "fields": {
    "title": "Mathematics",
    "slug": "mathematics"
  }
},
{
  "model": "courses.subject",
  "pk": 2,
  "fields": {
    "title": "Music",
    "slug": "music"
  }
},
{
  "model": "courses.subject",
  "pk": 3,
  "fields": {
    "title": "Physics",
    "slug": "physics"
  }
},
{
  "model": "courses.subject",
  "pk": 4,
  "fields": {
    "title": "Programming",
    "slug": "programming"
  }
}
]
```

The `dumpdata` command dumps data from the database into the standard output, serialized in JSON format by default. The resulting data structure includes information about the model and its fields for Django to be able to load it into the database.

You can limit the output to the models of an application by providing the application names to the command or specifying single models for outputting data using the `app.Model` format. You can also specify the format using the `--format` flag. By default, `dumpdata` outputs the serialized data to the standard output. However, you can indicate an output file using the `--output` flag. The `--indent` flag allows you to specify indentation. For more information on `dumpdata` parameters, run `python manage.py dumpdata --help`.

Save this dump to a fixtures file into a `fixtures/` directory in the `orders` application using the following commands:

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

Run the development server and use the administration site to remove the subjects you created. Then, load the fixture into the database using the following command:

```
python manage.py loaddata subjects.json
```

All `Subject` objects included in the fixture are loaded into the database.

By default, Django looks for files in the `fixtures/` directory of each application, but you can specify the complete path to the fixture file for the `loaddata` command. You can also use the `FIXTURE_DIRS` setting to tell Django additional directories to look for fixtures.

Fixtures are not only useful for setting up initial data, but also to provide sample data for your application or data required for your tests.

You can read about how to use fixtures for testing at `https://docs.djangoproject.com/en/2.0/topics/testing/tools/#fixture-loading`.

If you want to load fixtures in model migrations, take a look at Django's documentation about data migrations. You can find the documentation for migrating data at `https://docs.djangoproject.com/en/2.0/topics/migrations/#data-migrations`.

Creating models for diverse content

We plan to add different types of content to the course modules such as texts, images, files, and videos. We need a versatile data model that allows us to store diverse content. We are going to create a `Content` model that represents the modules' contents and define a generic relation to associate any kind of content.

Edit the `models.py` file of the `courses` application and add the following imports:

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

Then, add the following code to the end of the file:

```
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')
```

This is the `Content` model. A module contains multiple contents, so we define a `ForeignKey` field to the `Module` model. We also set up a generic relation to associate objects from different models that represent different types of content. Remember that we need three different fields to set up a generic relationship. In our `Content` model, these are:

- `content_type`: A `ForeignKey` field to the `ContentType` model
- `object_id`: This is `PositiveIntegerField` to store the primary key of the related object
- `item`: A `GenericForeignKey` field to the related object by combining the two previous fields

Only the `content_type` and `object_id` fields have a corresponding column in the database table of this model. The `item` field allows you to retrieve or set the related object directly, and its functionality is built on top of the other two fields.

We are going to use a different model for each type of content. Our content models will have some common fields, but they will differ in the actual data they can store.

Using model inheritance

Django supports model inheritance. It works in a similar way to standard class inheritance in Python. Django offers the following three options to use model inheritance:

- **Abstract models**: Useful when you want to put some common information into several models. No database table is created for the abstract model.
- **Multi-table model inheritance**: Applicable when each model in the hierarchy is considered a complete model by itself. A database table is created for each model.
- **Proxy models**: Useful when you need to change the behavior of a model, for example, by including additional methods, changing the default manager, or using different meta options. No database table is created for proxy models.

Let's take a closer look at each of them.

Abstract models

An **abstract model** is a base class in which you define fields you want to include in all child models. Django doesn't create any database table for abstract models. A database table is created for each child model, including the fields inherited from the abstract class and the ones defined in the child model.

To mark a model as abstract, you need to include `abstract=True` in its `Meta` class. Django will recognize that it is an abstract model and will not create a database table for it. To create child models, you just need to subclass the abstract model.

The following example shows an abstract `Content` model and a child `Text` model:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        abstract = True

class Text(BaseContent):
    body = models.TextField()
```

In this case, Django would create a table for the `Text` model only, including the `title`, `created`, and `body` fields.

Multi-table model inheritance

In multi-table inheritance, each model corresponds to a database table. Django creates a `OneToOneField` field for the relationship in the child's model to its parent.

To use multi-table inheritance, you have to subclass an existing model. Django will create a database table for both the original model and the sub-model. The following example shows multi-table inheritance:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django would include an automatically generated `OneToOneField` field in the `Text` model and create a database table for each model.

Proxy models

Proxy models are used to change the behavior of a model, for example, by including additional methods or different meta options. Both models operate on the database table of the original model. To create a proxy model, add `proxy=True` to the `Meta` class of the model.

The following example illustrates how to create a proxy model:

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

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class OrderedContent(BaseContent):
    class Meta:
        proxy = True
        ordering = ['created']

    def created_delta(self):
        return timezone.now() - self.created
```

Here, we define an `OrderedContent` model that is a proxy model for the `Content` model. This model provides a default ordering for QuerySets and an additional `created_delta()` method. Both models, `Content` and `OrderedContent`, operate on the same database table, and objects are accessible via the ORM through either model.

Creating the content models

The `Content` model of our `courses` application contains a generic relation to associate different types of content to it. We will create a different model for each type of content. All content models will have some fields in common and additional fields to store custom data. We are going to create an abstract model that provides the common fields for all content models.

Edit the `models.py` file of the `courses` application and add the following code to it:

```
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()
```

In this code, we define an abstract model named `ItemBase`. Therefore, we have set `abstract=True` in its `Meta` class. In this model, we define the `owner`, `title`, `created`, and `updated` fields. These common fields will be used for all types of content. The `owner` field allows us to store which user created the content. Since this field is defined in an abstract class, we need different `related_name` for each sub-model. Django allows us to specify a placeholder for the `model` class name in the `related_name` attribute as `%(class)s`. By doing so, `related_name` for each child model will be generated automatically. Since we use `'%(class)s_related'` as `related_name`, the reverse relation for child models will be `text_related`, `file_related`, `image_related`, and `video_related` respectively.

We have defined four different content models, which inherit from the `ItemBase` abstract model. These are as follows:

- `Text`: To store text content
- `File`: To store files, such as PDF
- `Image`: To store image files
- `Video`: To store videos; we use an `URLField` field to provide a video URL in order to embed it

Each child model contains the fields defined in the `ItemBase` class in addition to its own fields. A database table will be created for the `Text`, `File`, `Image`, and `Video` models respectively. There will be no database table associated to the `ItemBase` model since it is an abstract model.

Edit the `Content` model you created previously and modify its `content_type` field as follows:

```
content_type = models.ForeignKey(ContentType,
                    on_delete=models.CASCADE,
                    limit_choices_to={'model__in':(
                                      'text',
                                      'video',
                                      'image',
                                      'file')})
```

We add a `limit_choices_to` argument to limit the `ContentType` objects that can be used for the generic relationship. We use the `model__in` field lookup to filter the query to the `ContentType` objects with a `model` attribute that is `'text'`, `'video'`, `'image'`, or `'file'`.

Let's create a migration to include the new models we have added. Run the following command from the command line:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'courses':
  courses/migrations/0002_content_file_image_text_video.py
    - Create model Content
    - Create model File
    - Create model Image
    - Create model Text
    - Create model Video
```

Then, run the following command to apply the new migration:

```
python manage.py migrate
```

The output you see should end with the following line:

```
Applying courses.0002_content_file_image_text_video... OK
```

We have created models that are suitable to add diverse content to the course modules. However, there is still something missing in our models. The course modules and contents should follow a particular order. We need a field that allows us to order them easily.

Creating custom model fields

Django comes with a complete collection of model fields that you can use to build your models. However, you can also create your own model fields to store custom data or alter the behavior of existing fields.

We need a field that allows us to define an order for objects. An easy way to specify an order for objects using existing Django fields is by adding a `PositiveIntegerField` to your models. Using integers, we can easily specify the order of objects. We can create a custom order field that inherits from `PositiveIntegerField` and provides additional behavior.

There are two relevant functionalities that we will build into our order field:

- **Automatically assign an order value when no specific order is provided**: When saving a new object with no specific order, our field should automatically assign the number that comes after the last existing ordered object. If there are two objects with order `1` and `2` respectively, when saving a third object, we should automatically assign the order `3` to it if no specific order has been provided.
- **Order objects with respect to other fields**: Course modules will be ordered with respect to the course they belong to and module contents with respect to the module they belong to.

Create a new `fields.py` file inside the `courses` application directory and add the following code to it:

```
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(OrderField, self).__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):
        if getattr(model_instance, self.attname) is None:
            # no current value
            try:
                qs = self.model.objects.all()
                if self.for_fields:
                    # filter by objects with the same field values
                    # for the fields in "for_fields"
                    query = {field: getattr(model_instance, field)\
                    for field in self.for_fields}
                    qs = qs.filter(**query)
                # get the order of the last item
                last_item = qs.latest(self.attname)
                value = last_item.order + 1
            except ObjectDoesNotExist:
                value = 0
            setattr(model_instance, self.attname, value)
            return value
        else:
            return super(OrderField,
                         self).pre_save(model_instance, add)
```

This is our custom `OrderField`. It inherits from the `PositiveIntegerField` field provided by Django. Our `OrderField` field takes an optional `for_fields` parameter that allows us to indicate the fields that the order has to be calculated with respect to.

Our field overrides the `pre_save()` method of the `PositiveIntegerField` field, which is executed before saving the field into the database. In this method, we perform the following actions:

1. We check if a value already exists for this field in the model

instance. We use `self.attname`, which is the attribute name given to the field in the model. If the attribute's value is different than `None`, we calculate the order we should give it as follows:

1. We build a QuerySet to retrieve all objects for the field's model. We retrieve the model class the field belongs to by accessing `self.model`.
2. We filter the QuerySet by the fields' current value for the model fields that are defined in the `for_fields` parameter of the field, if any. By doing so, we calculate the order with respect to the given fields.

3. We retrieve the object with the highest order with `last_item = qs.latest(self.attname)` from the database. If no object is found, we assume this object is the first one and assign the order `0` to it.
4. If an object is found, we add `1` to the highest order found.
5. We assign the calculated order to the field's value in the model instance using `setattr()` and return it.

2. If the model instance has a value for the current field, we don't do anything.

When you create custom model fields, make them generic. Avoid hardcoding data that depends on a specific model or field. Your field should work in any model.

You can find more information about writing custom model fields at `https://docs.djangoproject.com/en/2.0/howto/custom-model-fields/`.

Adding ordering to module and content objects

Let's add the new field to our models. Edit the models.py file of the courses application, and import the OrderField class and a field to the Module model as follows:

```
from .fields import OrderField

class Module(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['course'])
```

We name the new field order, and we specify that the ordering is calculated with respect to the course by setting for_fields=['course']. This means that the order for a new module will be assigned adding 1 to the last module of the same Course object. Now, you can edit the __str__() method of the Module model to include its order as follows:

```
class Module(models.Model):
    # ...
    def __str__(self):
        return '{}. {}'.format(self.order, self.title)
```

Module contents also need to follow a particular order. Add an OrderField field to the Content model as follows:

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

This time, we specify that the order is calculated with respect to the module field. Finally, let's add a default ordering for both models. Add the following `Meta` class to the `Module` and `Content` models:

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']

class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

The `Module` and `Content` models should now look as follows:

```
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 '{}. {}'.format(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']
```

Let's create a new model migration that reflects the new order fields. Open the shell and run the following command:

```
python manage.py makemigrations courses
```

You will see the following output:

```
You are trying to add a non-nullable field 'order' to content without a
default; we can't do that (the database needs something to populate existing
rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a
null value for this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

Django is telling us that we have to provide a default value for the new `order` field for existing rows in the database. If the field had `null=True`, it would accept null values and Django would create the migration automatically instead of asking for a default value. We can specify a default value or cancel the migration and add a `default` attribute to the `order` field in the `models.py` file before creating the migration.

Enter `1` and press *Enter* to provide a default value for existing records. You will see the following output:

```
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do
e.g. timezone.now
Type 'exit' to exit this prompt
>>>
```

Enter `0` so that this is the default value for existing records and press *Enter*. Django will ask you for a default value for the `Module` model, too. Choose the first option and enter `0` as the default value again. Finally, you will see an output similar to the following one:

```
Migrations for 'courses':
  courses/migrations/0003_auto_20180326_0704.py
    - Change Meta options on content
    - Change Meta options on module
    - Add field order to content
    - Add field order to module
```

Then, apply the new migrations with the following command:

```
python manage.py migrate
```

The output of the command will inform you that the migration was successfully applied, as follows:

```
Applying courses.0003_auto_20180326_0704... OK
```

Let's test our new field. Open the shell with the following command:

```
python manage.py shell
```

Create a new course as follows:

```
>>> from django.contrib.auth.models import User
>>> from courses.models import Subject, Course, Module
>>> user = User.objects.last()
>>> subject = Subject.objects.last()
>>> c1 = Course.objects.create(subject=subject, owner=user, title='Course 1',
slug='course1')
```

We have created a course in the database. Now, let's add modules to the course and see how their order is automatically calculated. We create an initial module and check its order:

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

`OrderField` sets its value to `0`, since this is the first `Module` object created for the given course. Now, we create a second module for the same course:

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

`OrderField` calculates the next order value adding `1` to the highest order for existing objects. Let's create a third module, forcing a specific order:

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

If we specify a custom order, the `OrderField` field does not interfere and the value given to the `order` is used.

Let's add a fourth module:

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

The order for this module has been automatically set. Our `OrderField` field does not guarantee that all order values are consecutive. However, it respects existing order values and always assigns the next order based on the highest existing order.

Let's create a second course and add a module to it:

```
>>> c2 = Course.objects.create(subject=subject, title='Course 2',
slug='course2', owner=user)
>>> m5 = Module.objects.create(course=c2, title='Module 1')
>>> m5.order
0
```

To calculate the new module's order, the field only takes into consideration existing modules that belong to the same course. Since this is the first module of the second course, the resulting order is `0`. This is because we specified `for_fields=['course']` in the `order` field of the `Module` model.

Congratulations! You have successfully created your first custom model field.

Creating a CMS

Now that we have created a versatile data model, we are going to build the CMS. The CMS will allow instructors to create courses and manage their contents. We need to provide the following functionality:

- Log in to the CMS
- List the courses created by the instructor
- Create, edit, and delete courses
- Add modules to a course and reorder them
- Add different types of content to each module and reorder contents

Adding an authentication system

We are going to use Django's authentication framework in our platform. Both instructors and students will be instances of Django's `User` model, so they will be able to log in to the site using the authentication views of `django.contrib.auth`.

Edit the main `urls.py` file of the `educa` project and include the `login` and `logout` views of Django's authentication framework:

```
from django.contrib import admin
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('admin/', admin.site.urls),
]
```

Creating the authentication templates

Create the following file structure inside the `courses` application directory:

```
templates/
    base.html
    registration/
        login.html
        logged_out.html
```

Before building the authentication templates, we need to prepare the base template for our project. Edit the `base.html` template file and add the following content to it:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>{% block title %}Educa{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <a href="/" class="logo">Educa</a>
    <ul class="menu">
      {% if request.user.is_authenticated %}
        <li><a href="{% url "logout" %}">Sign out</a></li>
      {% else %}
        <li><a href="{% url "login" %}">Sign in</a></li>
      {% endif %}
    </ul>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>

  <script src="https://ajax.googleapis.com/ajax/libs/jquery/
   3.3.1/jquery.min.js"></script>
  <script>
    $(document).ready(function() {
      {% block domready %}
      {% endblock %}
    });
  </script>
</body>
</html>
```

This is the base template that will be extended by the rest of the templates. In this template, we define the following blocks:

- `title`: The block for other templates to add a custom title for each page.
- `content`: The main block for content. All templates that extend the base template should add content to this block.
- `domready`: Located inside the `$document.ready()` function of jQuery. It allows us to execute code when the DOM has finished loading.

The CSS styles used in this template are located in the `static/` directory of the `courses` application, in the code that comes along with this chapter. Copy the `static/` directory into the same directory of your project to use them.

Edit the `registration/login.html` template and add the following code to it:

```
{% 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 %}
```

This is a standard login template for Django's `login` view.

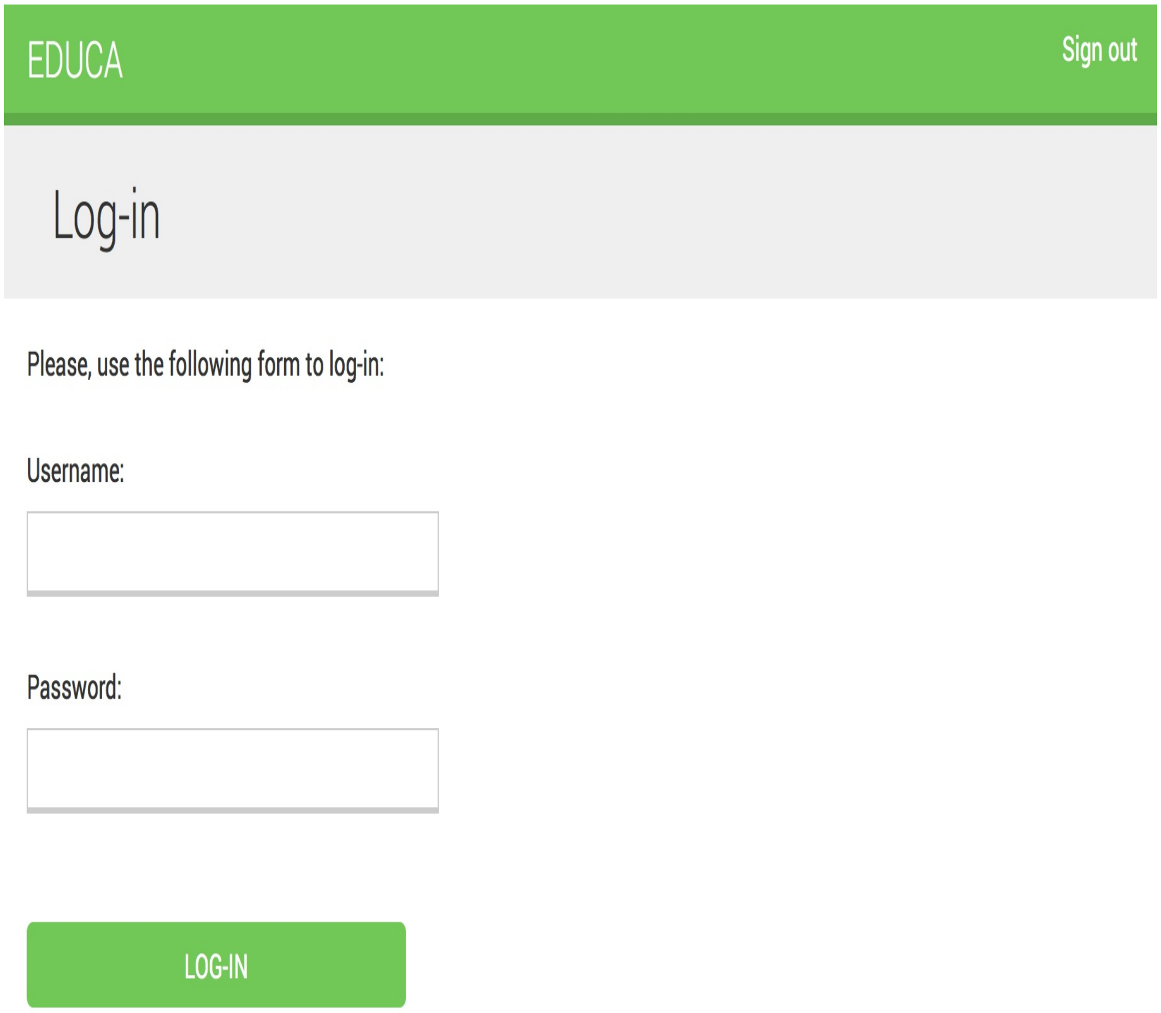
Edit the `registration/logged_out.html` template and add the following code to it:

```
{% 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 %}
```

This is the template that will be displayed to the user after logout. Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/accounts/login/` in your browser. You should see the login page like this:


EDUCA
Sign out
Log-in
Please, use the following form to log-in:
Username:
Password:
LOG-IN

Creating class-based views

We are going to build views to create, edit, and delete courses. We will use class-based views for this. Edit the `views.py` file of the `courses` application and add the following code to it:

```
from django.views.generic.list import ListView
from .models import Course

class ManageCourseListView(ListView):
    model = Course
    template_name = 'courses/manage/course/list.html'

    def get_queryset(self):
        qs = super(ManageCourseListView, self).get_queryset()
        return qs.filter(owner=self.request.user)
```

This is the `ManageCourseListView` view. It inherits from Django's generic `ListView`. We override the `get_queryset()` method of the view to retrieve only courses created by the current user. To prevent users from editing, updating, or deleting courses they didn't create, we will also need to override the `get_queryset()` method in the create, update, and delete views. When you need to provide a specific behavior for several class-based views, it is recommended to use *mixins*.

Using mixins for class-based views

Mixins are a special kind of multiple inheritance for a class. You can use them to provide common discrete functionality that, added to other mixins, allows you to define the behavior of a class. There are two main situations to use mixins:

- You want to provide multiple optional features for a class
- You want to use a particular feature in several classes

Django comes with several mixins that provide additional functionality to your class-based views. You can learn more about mixins at `https://docs.djangoproject.com/en/2.0/topics/class-based-views/mixins/`.

We are going to create a mixin class that includes a common behavior and use it for the course's views. Edit the `views.py` file of the `courses` application and modify it as follows:

```
from django.urls import reverse_lazy
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, UpdateView, \
                                      DeleteView
from .models import Course

class OwnerMixin(object):
    def get_queryset(self):
        qs = super(OwnerMixin, self).get_queryset()
        return qs.filter(owner=self.request.user)

class OwnerEditMixin(object):
    def form_valid(self, form):
        form.instance.owner = self.request.user
        return super(OwnerEditMixin, self).form_valid(form)

class OwnerCourseMixin(OwnerMixin):
    model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
    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'
    success_url = reverse_lazy('manage_course_list')
```

In this code, we create the `OwnerMixin` and `OwnerEditMixin` mixins. We will use these mixins together with the `ListView`, `CreateView`, `UpdateView`, and `DeleteView` views provided by Django. `OwnerMixin` implements the following method:

- `get_queryset()`: This method is used by the views to get the base QuerySet. Our mixin will override this method to filter objects by the `owner` attribute to retrieve objects that belong to the current user (`request.user`).

`OwnerEditMixin` implements the following method:

- `form_valid()`: This method is used by views that use Django's `ModelFormMixin` mixin, that is, views with forms or modelforms such as `CreateView` and `UpdateView`. `form_valid()` are executed when the submitted form is valid. The default behavior for this method is saving the instance (for modelforms) and redirecting the user to `success_url`. We override this method to automatically set the current user in the `owner` attribute of the object being saved. By doing so, we set the owner for an object automatically when it is saved.

Our `OwnerMixin` class can be used for views that interact with any model that contains an `owner` attribute.

We also define an `OwnerCourseMixin` class that inherits `OwnerMixin` and provides the following attribute for child views:

- `model`: The model used for QuerySets. Used by all views.

We define a `OwnerCourseEditMixin` mixin with the following attributes:

- `fields`: The fields of the model to build the model form of the `CreateView` and `UpdateView` views.
- `success_url`: Used by `CreateView` and `UpdateView` to redirect the user after the form is successfully submitted. We use a URL with the name `manage_course_list` that we are going to create later.

Finally, we create the following views that subclass `OwnerCourseMixin`:

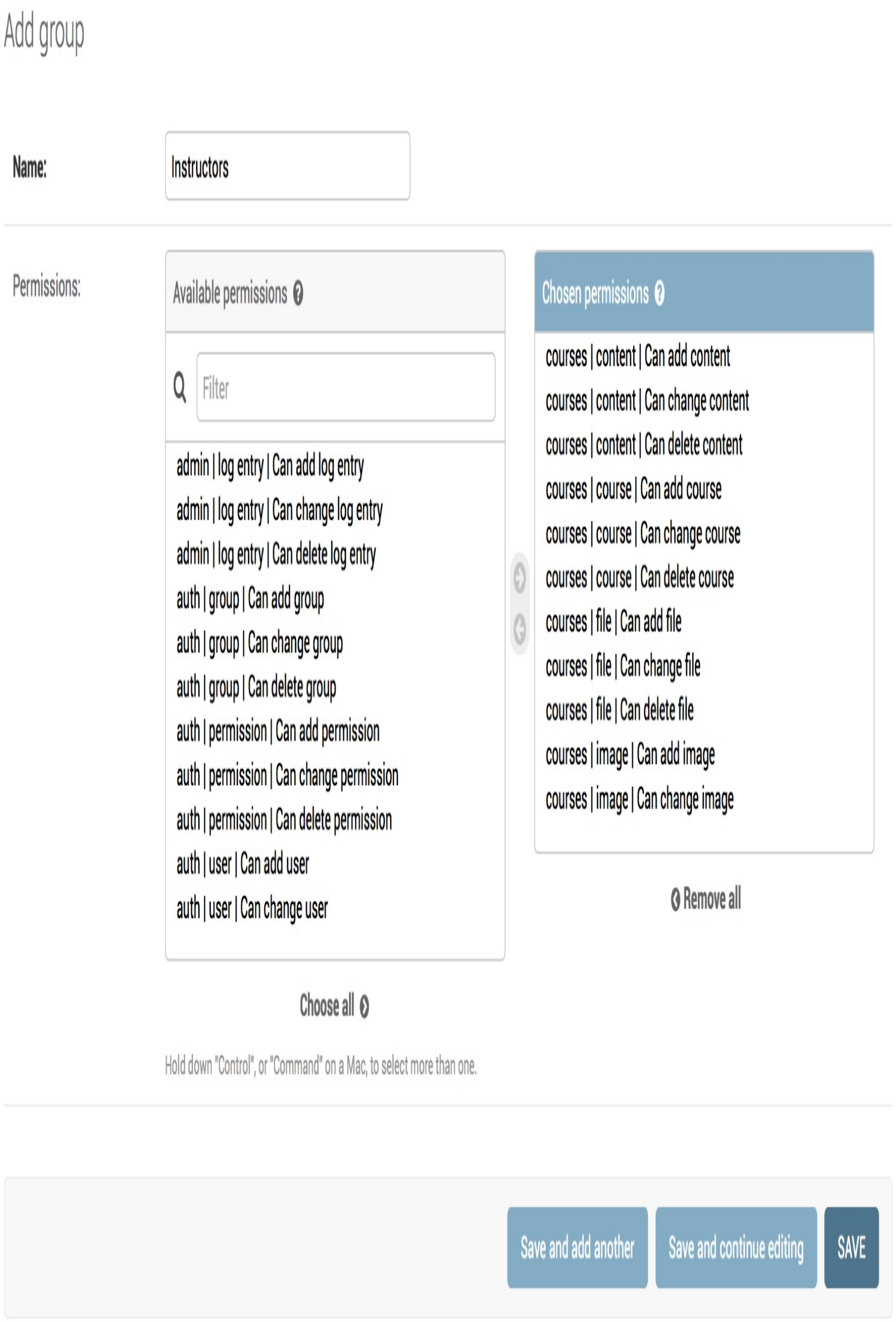
- `ManageCourseListView`: Lists the courses created by the user. It inherits from `OwnerCourseMixin` and `ListView`.
- `CourseCreateView`: Uses a modelform to create a new `Course` object. It uses the fields defined in `OwnerCourseEditMixin` to build a model form and also subclasses `CreateView`.

- `CourseUpdateView`: Allows editing an existing `Course` object. It inherits from `OwnerCourseEditMixin` and `UpdateView`.
- `CourseDeleteView`: Inherits from `OwnerCourseMixin` and the generic `DeleteView`. Defines `success_url` to redirect the user after the object is deleted.

Working with groups and permissions

We have created the basic views to manage courses. Currently, any user could access these views. We want to restrict these views so that only instructors have permission to create and manage courses. Django's authentication framework includes a permission system that allows you to assign permissions to users and groups. We are going to create a group for instructor users and assign permissions to create, update, and delete courses.

Run the development server using the command and open `http://127.0.0.1:8000/admin/auth/group/add/` in your browser to create a new `Group` object. Add the name `Instructors` and choose all permissions of the `courses` application except those of the `Subject` model, as follows:



Add group

Name: Instructors

Permissions:

Available permissions

Filter

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | user | Can add user
auth | user | Can change user

Choose all

Chosen permissions

courses | content | Can add content
courses | content | Can change content
courses | content | Can delete content
courses | course | Can add course
courses | course | Can change course
courses | course | Can delete course
courses | file | Can add file
courses | file | Can change file
courses | file | Can delete file
courses | image | Can add image
courses | image | Can change image

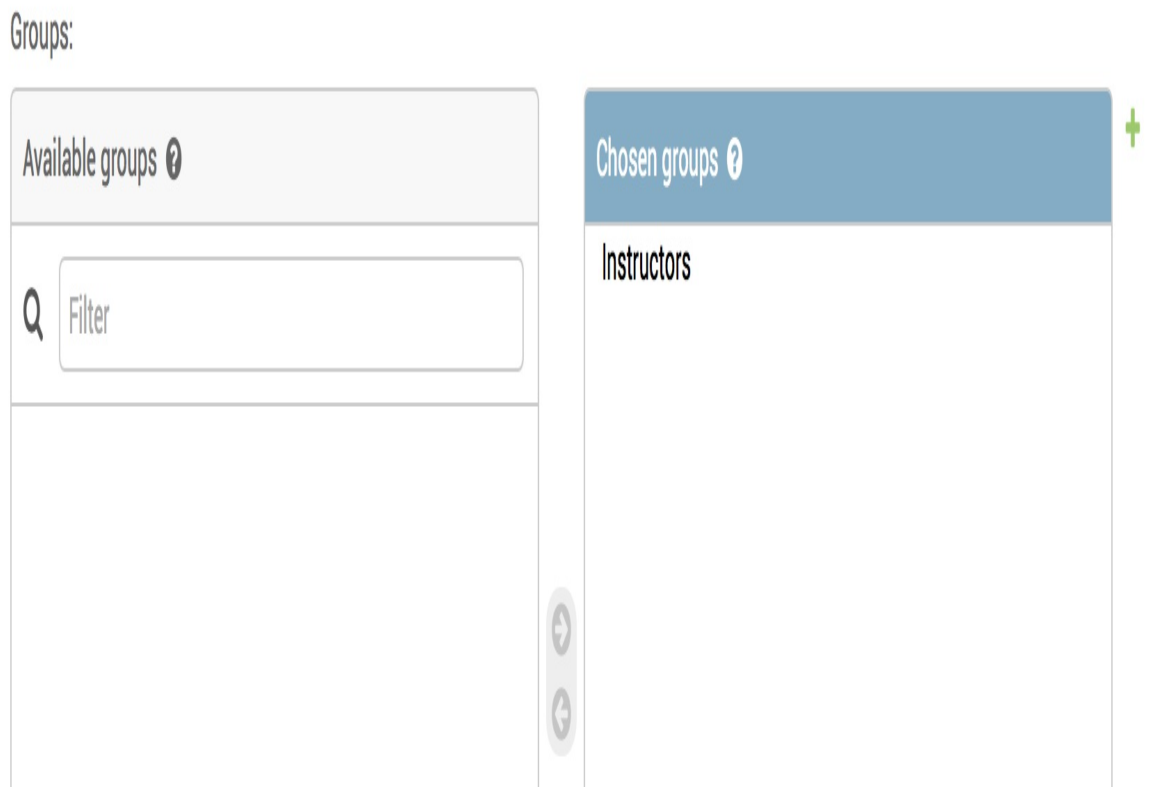
Remove all

Hold down "Control", or "Command" on a Mac, to select more than one.

As you can see, there are three different permissions for each model: *can add*, *can change*, and *can delete*. After choosing permissions for this group, click on the SAVE button.

Django creates permissions for models automatically, but you can also create custom permissions. You can read more about adding custom permissions at `https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#custom-permissions`.

Open `http://127.0.0.1:8000/admin/auth/user/add/` and create a new user. Edit the user and add it to the Instructors group, as follows:



Users inherit the permissions of the groups they belong to, but you can also add individual permissions to a single user using the administration site. Users that have `is_superuser` set to `True` have all permissions automatically.

Restricting access to class-based views

We are going to restrict access to the views so that only users with the appropriate permissions can add, change, or delete `Course` objects. We are going to use the following two mixins provided by `django.contrib.auth` to limit access to views:

- `LoginRequiredMixin`: Replicates the `login_required` decorator's functionality.
- `PermissionRequiredMixin`: Grants access to the view to users that have a specific permission. Remember that superusers automatically have all permissions.

Edit the `views.py` file of the `courses` application and add the following import:

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

Make `OwnerCourseMixin` inherit `LoginRequiredMixin` like this:

```
class OwnerCourseMixin(OwnerMixin, LoginRequiredMixin):
    model = Course
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

Then, add a `permission_required` attribute to the create, update, and delete views, as follows:

```
class CourseCreateView(PermissionRequiredMixin,
                       OwnerCourseEditMixin,
                       CreateView):
    permission_required = 'courses.add_course'

class CourseUpdateView(PermissionRequiredMixin,
                       OwnerCourseEditMixin,
                       UpdateView):
    permission_required = 'courses.change_course'

class CourseDeleteView(PermissionRequiredMixin,
                       OwnerCourseMixin,
                       DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
    permission_required = 'courses.delete_course'
```

`PermissionRequiredMixin` checks that the user accessing the view has the permission specified in the `permission_required` attribute. Our views are now only accessible to users that have proper permissions.

Let's create URLs for these views. Create a new file inside the `courses` application directory and name it `urls.py`. Add the following code to it:

```
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'),
]
```

These are the URL patterns for the list, create, edit, and delete course views. Edit the main `urls.py` file of the `educa` project and include the URL patterns of the `courses` application, as follows:

```
from django.urls import path, include

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('admin/', admin.site.urls),
    path('course/', include('courses.urls')),
]
```

We need to create the templates for these views. Create the following directories and files inside the `templates/` directory of the `courses` application:

```
courses/
    manage/
        course/
            list.html
            form.html
            delete.html
```

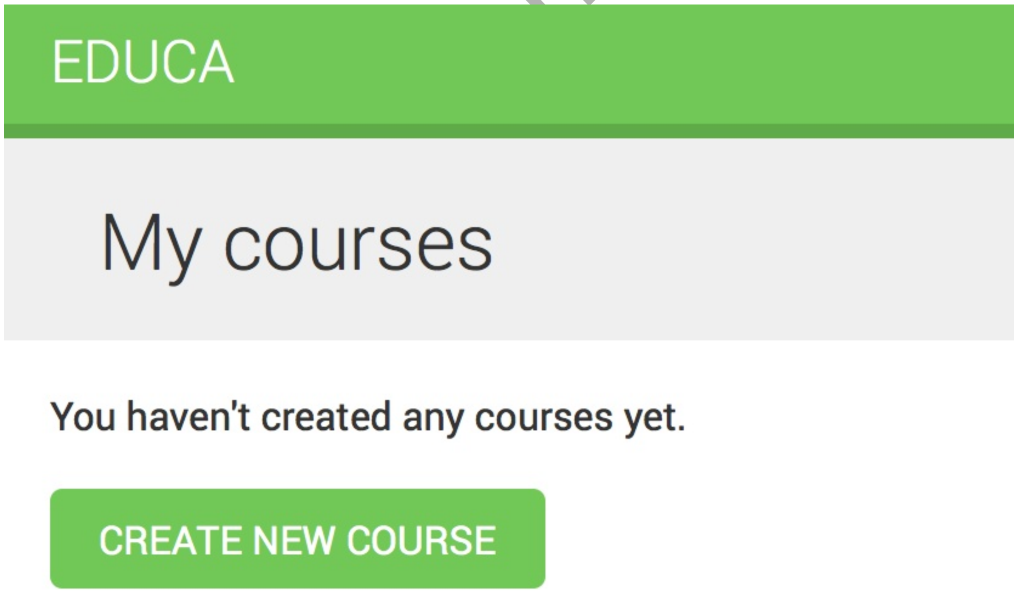
Edit the `courses/manage/course/list.html` template and add the following code to it:

```
{% 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 %}
```

This is the template for the `ManageCourseListView` view. In this template, we list the courses created by the current user. We include links to edit or delete each course, and a link to create new courses.

Run the development server using the command `python manage.py runserver`. Open `http://127.0.0.1:8000/accounts/login/?next=/course/mine/` in your browser and log in with a user that belongs to the `Instructors` group. After logging in, you will be redirected to the `http://127.0.0.1:8000/course/mine/` URL and you should see the following page:



You haven't created any courses yet.

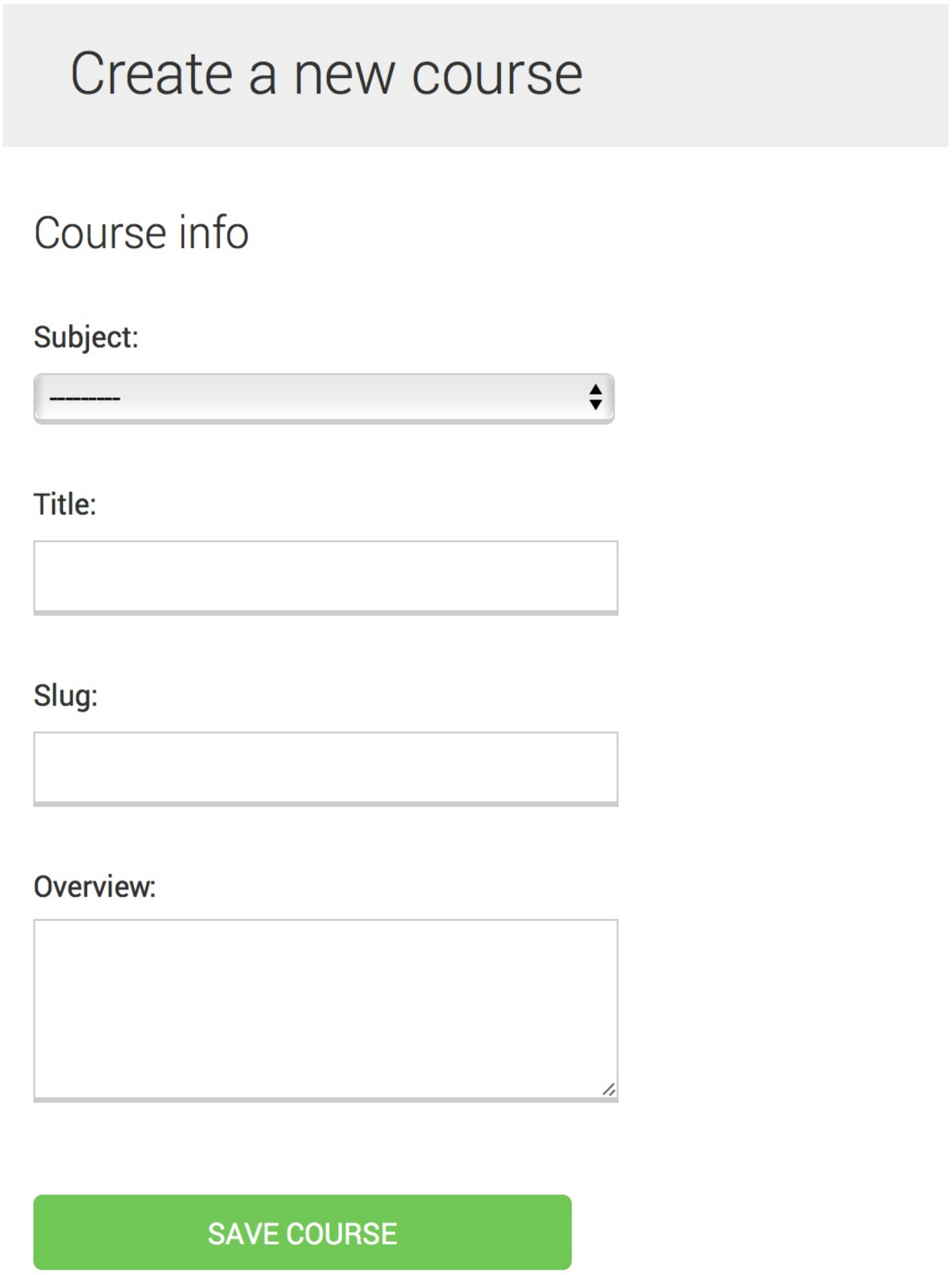
This page will display all courses created by the current user.

Let's create the template that displays the form for the create and update course views. Edit the `courses/manage/course/form.html` template and write the following code:

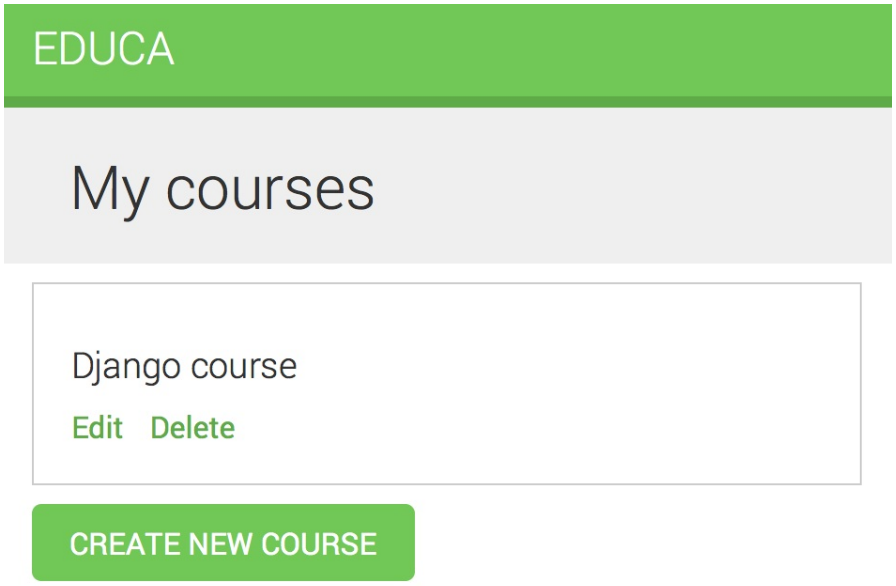
```
{% extends "base.html" %}

{% block title %}
  {% if object %}
    Edit course "{{ object.title }}"
  {% else %}
    Create a new course
  {% endif %}
{% endblock %}

{% block content %}
  <h1>
    {% if object %}
      Edit course "{{ object.title }}"
    {% else %}
      Create a new course
    {% endif %}
  </h1>
  <div class="module">
    <h2>Course info</h2>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Save course"></p>
    </form>
  </div>
{% endblock %}
```

The `form.html` template is used for both the `CourseCreateView` and `CourseUpdateView` views. In this template, we check whether an `object` variable is in the context. If `object` exists in the context, we know that we are updating an existing course, and we use it in the page title. Otherwise, we are creating a new `Course` object.

Open `http://127.0.0.1:8000/course/mine/` in your browser and click the CREATE NEW COURSE button. You will see the following page:



Fill in the form and click the SAVE COURSE button. The course will be saved and you will be redirected to the course list page. It should look as follows:



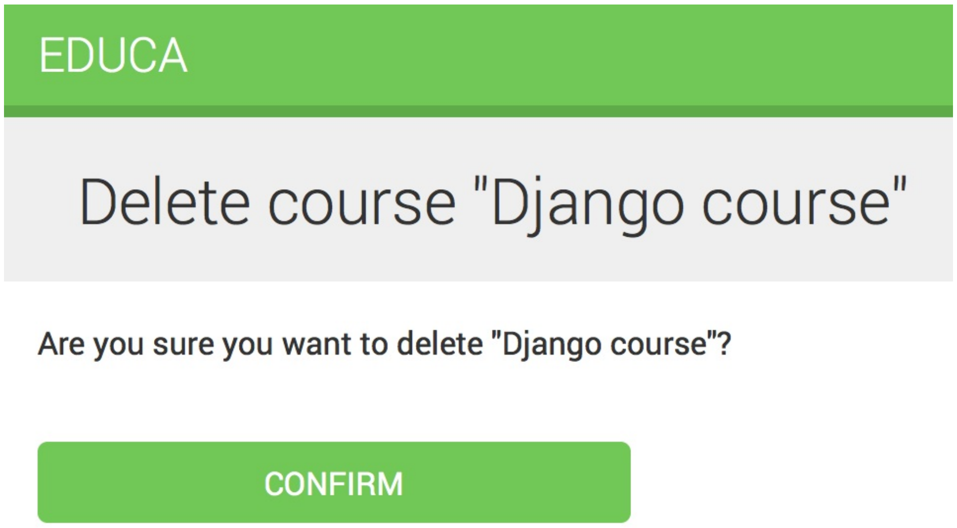
Then, click the Edit link for the course you have just created. You will see the form again, but this time you are editing an existing `Course` object instead of creating one.

Finally, edit the `courses/manage/course/delete.html` template and add the following code:

```
{% extends "base.html" %}

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

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

  <div class="module">
    <form action="" method="post">
      {% csrf_token %}
      <p>Are you sure you want to delete "{{ object }}"?</p>
      <input type="submit" class"button" value="Confirm">
    </form>
  </div>
{% endblock %}
```

This is the template for the `CourseDeleteView` view. This view inherits from `DeleteView` provided by Django, which expects user confirmation to delete an object.

Open your browser and click the Delete link of your course. You should see the following confirmation page:


EDUCA
Delete course "Django course"
Are you sure you want to delete "Django course"?
CONFIRM

Click the CONFIRM button. The course will be deleted and you will be redirected to the course list page again.

Instructors can now create, edit, and delete courses. Next, we need to provide them with CMS to add modules and contents to courses. We will start by managing course modules.

Managing course modules and content

We are going to build a system to manage course modules and their contents. We will need to build forms that can be used for managing multiple modules per course and different types of content for each module. Both modules and contents will have to follow a specific order and we should be able to reorder them using the CMS.

Using formsets for course modules

Django comes with an abstraction layer to work with multiple forms on the same page. These groups of forms are known as *formsets*. Formsets manage multiple instances of a certain `Form` or `ModelForm`. All forms are submitted at once and the formset takes care of the initial number of forms to display, limiting the maximum number of forms that can be submitted and validating all the forms.

Formsets include an `is_valid()` method to validate all forms at once. You can also provide initial data for the forms and specify how many additional empty forms to display.

You can learn more about formsets at `https://docs.djangoproject.com/en/2.0/topics/forms/formsets/` and about model formsets at `https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/#model-formsets`.

Since a course is divided into a variable number of modules, it makes sense to use formsets to manage them. Create a `forms.py` file in the `courses` application directory and add the following code to it:

```
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)
```

This is the `ModuleFormSet` formset. We build it using the `inlineformset_factory()` function provided by Django. Inline formsets are a small abstraction on top of formsets that simplify working with related objects. This function allows us to build a model formset dynamically for the `Module` objects related to a `Course` object.

We use the following parameters to build the formset:

- `fields`: The fields that will be included in each form of the formset.
- `extra`: Allows us to set the number of empty extra forms to display in the formset.
- `can_delete`: If you set this to `True`, Django will include a Boolean field for each form that will be rendered as a checkbox input. It allows you to mark the objects you want to delete.

Edit the `views.py` file of the `courses` application and add the following code to it:

```
from django.shortcuts import redirect, get_object_or_404
from django.views.generic.base import TemplateResponseMixin, View
from .forms import ModuleFormSet

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(CourseModuleUpdateView,
                     self).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})
```

The `CourseModuleUpdateView` view handles the formset to add, update, and delete modules for a specific course. This view inherits from the following mixins and views:

- `TemplateResponseMixin`: This mixin takes charge of rendering templates and returning an HTTP response. It requires a `template_name` attribute that indicates the template to be rendered and provides the `render_to_response()` method to pass it a context and render the template.
- `View`: The basic class-based view provided by Django.

In this view, we implement the following methods:

- `get_formset()`: We define this method to avoid repeating the code to build the formset. We create a `ModuleFormSet` object for the given `Course` object with optional data.
- `dispatch()`: This method is provided by the `View` class. It takes an HTTP request and its parameters and attempts to delegate to a lowercase method that matches the HTTP method used: a `GET` request is delegated to the `get()` method and a `POST` request to `post()`, respectively. In this method, we use the `get_object_or_404()` shortcut function to get the `Course`

object for the given id parameter that belongs to the current user. We include this code in the dispatch() method because we need to retrieve the course for both GET and POST requests. We save it into the course attribute of the view to make it accessible to other methods.

- get(): Executed for GET requests. We build an empty ModuleFormSet formset and render it to the template together with the current Course object using the render_to_response() method provided by TemplateResponseMixin.
- post(): Executed for POST requests. In this method, we perform the following actions:

1. We build a ModuleFormSet instance using the submitted data.
2. We execute the is_valid() method of the formset to validate all of its forms.
3. If the formset is valid, we save it by calling the save() method. At this point, any changes made, such as adding, updating, or marking modules for deletion, are applied to the database. Then, we redirect users to the manage_course_list URL. If the formset is not valid, we render the template to display any errors, instead.

Edit the urls.py file of the courses application and add the following URL pattern to it:

```
path('<pk>/module/',
     views.CourseModuleUpdateView.as_view(),
     name='course_module_update'),
```

Create a new directory inside the courses/manage/ template directory and name it `module`. Create a `courses/manage/module/formset.html` template and add the following code to it:

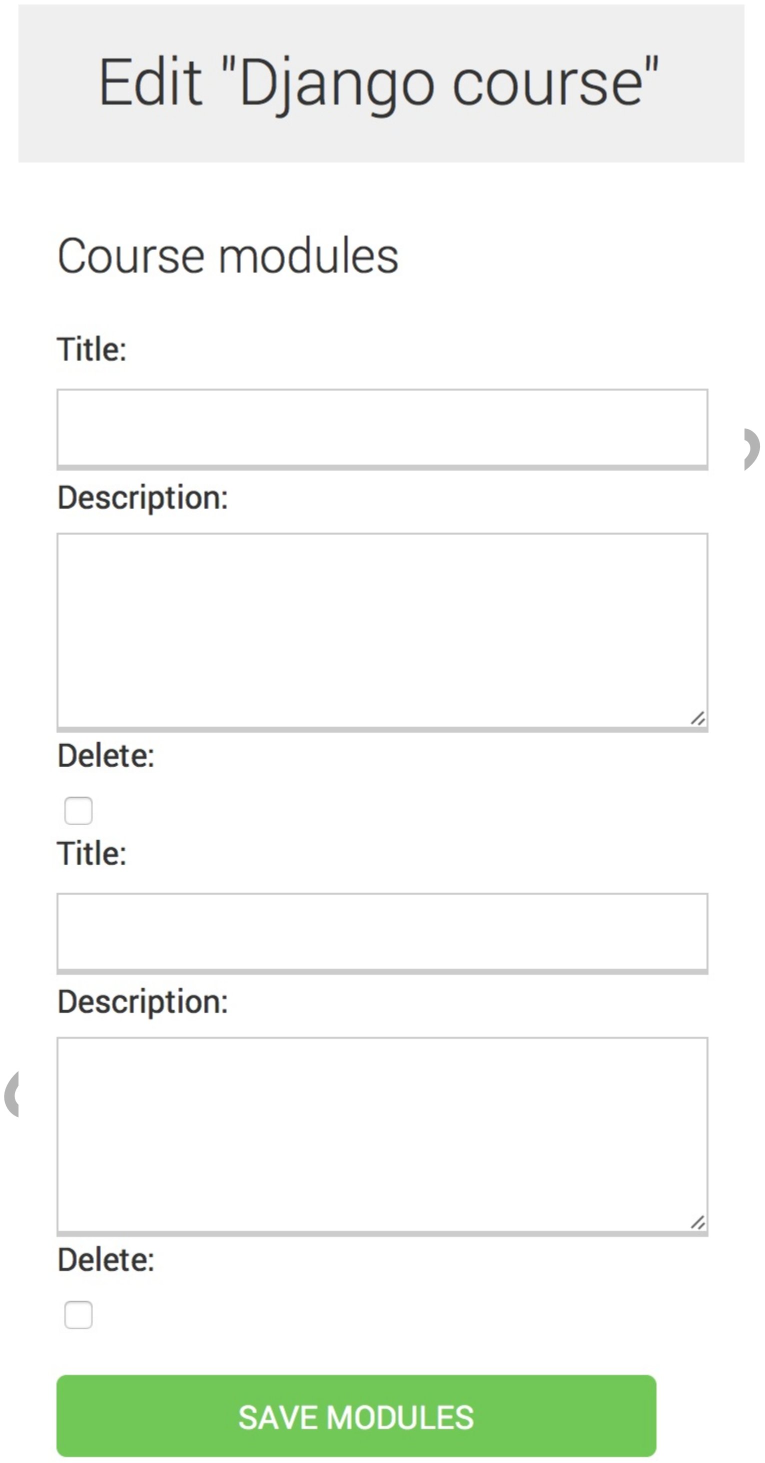
```
{% extends "base.html" %}

{% block title %}
  Edit "{{ course.title }}"
{% endblock %}

{% block content %}
  <h1>Edit "{{ course.title }}"</h1>
  <div class="module">
    <h2>Course modules</h2>
    <form action="" method="post">
      {{ formset }}
      {{ formset.management_form }}
      {% csrf_token %}
      <input type="submit" class="button" value="Save modules">
    </form>
  </div>
{% endblock %}
```

In this template, we create a `<form>` HTML element, in which we include `formset`. We also include the management form for the formset with the variable `{{ formset.management_form }}`. The management form includes hidden fields to control the initial, total, minimum, and maximum number of forms. You can see it's very easy to create a formset.

Edit the `courses/manage/course/list.html` template and add the following link for the `course_module_update` URL below the course edit and delete links:

```
<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>
```

We have included the link to edit the course modules. Open `http://127.0.0.1:8000/course/mine/` in your browser. Create a course and click the Edit modules link for it. You should see a formset as follows:



The formset includes a form for each `Module` object contained in the course. After these, two empty extra forms are displayed because we set `extra=2` for `ModuleFormSet`. When you save the formset, Django will include another two extra fields to add new modules.

Adding content to course modules

Now, we need a way to add content to course modules. We have four different types of content: text, video, image, and file. We can consider creating four different views to create content, one for each model. Yet we are going to take a more generic approach and create a view that handles creating or updating objects of any content model.

Edit the `views.py` file of the `courses` application and add the following code to it:

```
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(ContentCreateUpdateView,
           self).dispatch(request, module_id, model_name, id)
```

This is the first part of `ContentCreateUpdateView`. It will allow us to create and update contents of different models. This view defines the following methods:

- `get_model()`: Here, we check that the given model name is one of the four content models: `Text`, `Video`, `Image`, or `File`. Then, we use Django's `apps module` to obtain the actual class for the given model name. If the given model name is not one of the valid ones, we return `None`.
- `get_form()`: We build a dynamic form using the `modelform_factory()` function of the form's framework. Since we are going to build a form for the `Text`, `Video`, `Image`, and `File` models, we use the `exclude` parameter to specify the common fields to exclude from the form and let all other attributes be included automatically. By doing so, we don't have to know which fields to include depending on the model.
- `dispatch()`: It receives the following URL parameters and stores the corresponding module, model, and content object as class attributes:
    - `module_id`: The ID for the module that the content is/will be associated with.

- `model_name`: The model name of the content to create/update.
- `id`: The ID of the object that is being updated. It's `None` to create new objects.

Add the following `get()` and `post()` methods to `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})
```

These methods are as follows:

- `get()`: Executed when a `GET` request is received. We build the model form for the `Text`, `Video`, `Image`, or `File` instance that is being updated. Otherwise, we pass no instance to create a new object, since `self.obj` is `None` if no ID is provided.
- `post()`: Executed when a `POST` request is received. We build the

modelform passing any submitted data and files to it. Then, we validate it. If the form is valid, we create a new object and assign `request.user` as its owner before saving it to the database. We check for the `id` parameter. If no ID is provided, we know the user is creating a new object instead of updating an existing one. If this is a new object, we create a `Content` object for the given module and associate the new content to it.

Edit the `urls.py` file of the `courses` application and add the following URL patterns to it:

```
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'),
```

The new URL patterns are as follows:

- `module_content_create`: To create new text, video, image, or file objects and add them to a module. It includes the `module_id` and `model_name` parameters. The first one allows linking the new content object to the given module. The latter specifies the content model to build the form for.
- `module_content_update`: To update an existing text, video, image, or file object. It includes the `module_id` and `model_name` parameters and an `id` parameter to identify the content that is being updated.

Create a new directory inside the `courses/manage/` template directory and name it `content`. Create the template `courses/manage/content/form.html` and add the following code to it:

```
{% extends "base.html" %}

{% block title %}
  {% if object %}
    Edit content "{{ object.title }}"
  {% else %}
    Add a new content
  {% endif %}
{% endblock %}

{% block content %}
  <h1>
    {% if object %}
      Edit content "{{ object.title }}"
    {% else %}
      Add a new content
    {% endif %}
  </h1>
  <div class="module">
    <h2>Course info</h2>
    <form action="" method="post" enctype="multipart/form-data">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Save content"></p>
    </form>
  </div>
{% endblock %}
```

This is the template for the `ContentCreateUpdateView` view. In this template, we check whether an `object` variable is in the context. If `object` exists in the context, we are updating an existing object. Otherwise, we are creating a new object.

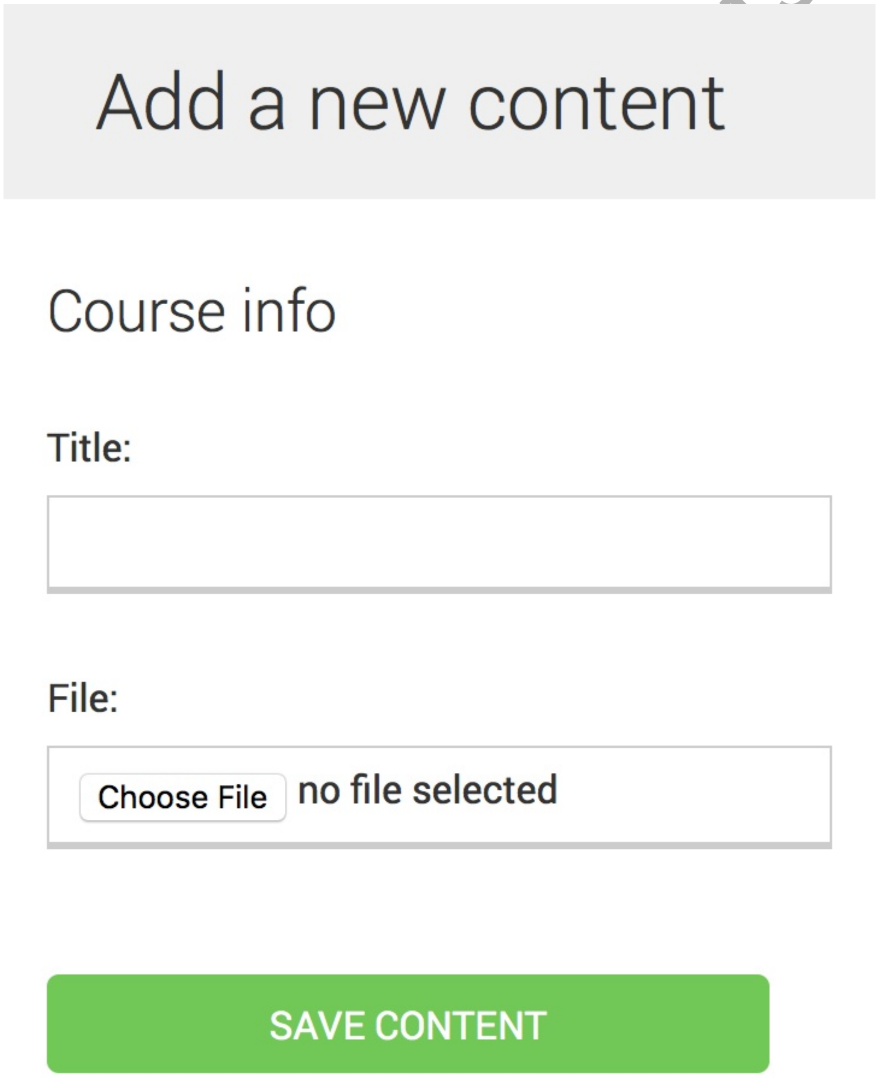
We include `enctype="multipart/form-data"` in the `<form>` HTML element; because the form contains a file upload for the `File` and `Image` content models.

Run the development server, open `http://127.0.0.1:8000/course/mine/`, click Edit modules for an existing course, and create a module.

Open the Python shell with the command `python manage.py shell` and obtain the ID of the most recently created module, as follows:

```
>>> from courses.models import Module
>>> Module.objects.latest('id').id
6
```

Run the development server and open `http://127.0.0.1:8000/course/module/6/content/image/create/` in your browser, replacing the module ID by the one you obtained before. You will see the form to create an `Image` object, as follows:


Add a new content

Course info

Title:

File:

Don't submit the form yet. If you try to do so, it will fail because we haven't defined the module_content_list URL yet. We are going to create it in a bit.

We also need a view for deleting contents. Edit the views.py file of the courses application and add the following code:

```
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)
```

The ContentDeleteView class retrieves the Content object with the given ID; it deletes the related Text, Video, Image, or File object; and finally, it deletes the Content object and redirects the user to the module_content_list URL to list the other contents of the module.

Edit the urls.py file of the courses application and add the following URL pattern to it:

```
path('content/<int:id>/delete/',
     views.ContentDeleteView.as_view(),
     name='module_content_delete'),
```

Now, instructors can create, update, and delete contents easily.

Managing modules and contents

We have built views to create, edit, and delete course modules and contents. Now, we need a view to display all modules for a course and list contents for a specific module.

Edit the `views.py` file of the `courses` application and add the following code to it:

```
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})
```

This is `ModuleContentListView`. This view gets the `Module` object with the given ID that belongs to the current user and renders a template with the given module.

Edit the `urls.py` file of the `courses` application and add the following URL pattern to it:

```
path('module/<int:module_id>/',
     views.ModuleContentListView.as_view(),
     name='module_content_list'),
```

Create a new template inside the `templates/courses/manage/module/` directory and name it `content_list.html`. Add the following code to it:

```
{% 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 %}
```

This is the template that displays all modules for a course and the contents of the selected module. We iterate over the course modules to display them in a sidebar. We iterate over the module's contents and access `content.item` to get the related `Text`, `Video`, `Image`, or `File` object. We also include links to create new text, video, image, or file contents.

We want to know which type of object each of the `item` objects is: `Text`, `Video`, `Image`, or `File`. We need the model name to build the URL to edit the object. Besides this, we could display each item in the template differently, based on the type of content it is. We can get the model for an object from the model's `Meta` class, by accessing the object's `_meta` attribute. Nevertheless, Django doesn't allow accessing variables or attributes starting with an underscore in templates to prevent retrieving private attributes or calling private methods. We can solve this by writing a custom template filter.

Create the following file structure inside the `courses` application directory:

```
templatetags/
    __init__.py
    course.py
```

Edit the `course.py` module and add the following code to it:

```
from django import template

register = template.Library()

@register.filter
def model_name(obj):
    try:
        return obj._meta.model_name
    except AttributeError:
        return None
```

This is the `model_name` template filter. We can apply it in templates as `object|model_name` to get the model name for an object.

Edit the `templates/courses/manage/module/content_list.html` template and add the following line below the `{% extends %}` template tag:

```
{% load course %}
```

This will load the `course` template tags. Then, replace the following lines:

```
<p>{{ item }}</p>
<a href="#">Edit</a>
```

Replace them with the following ones:

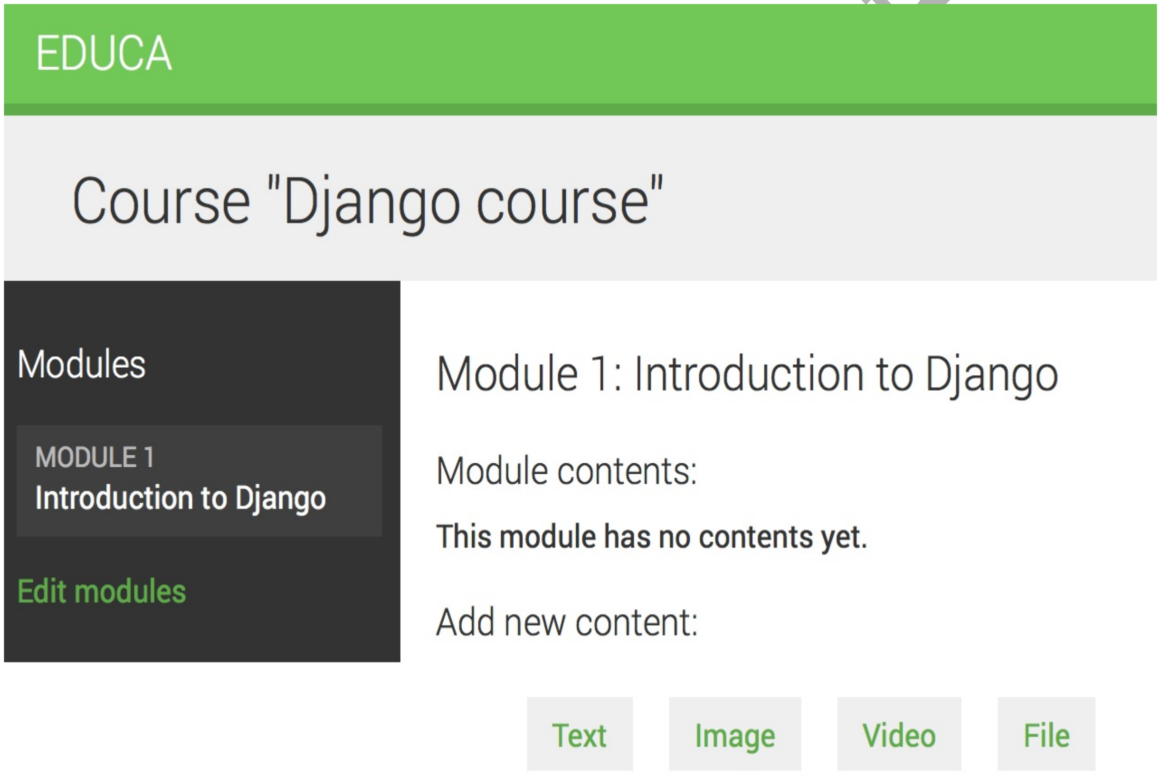
```
<p>{{ item }} ({{ item|model_name }})</p>
<a href="{% url "module_content_update" module.id item|model_name item.id
%}">Edit</a>
```

Now, we display the item model in the template and use the model name to build the link to edit the object. Edit the `courses/manage/course/list.html` template and add a link to the `module_content_list` URL like this:

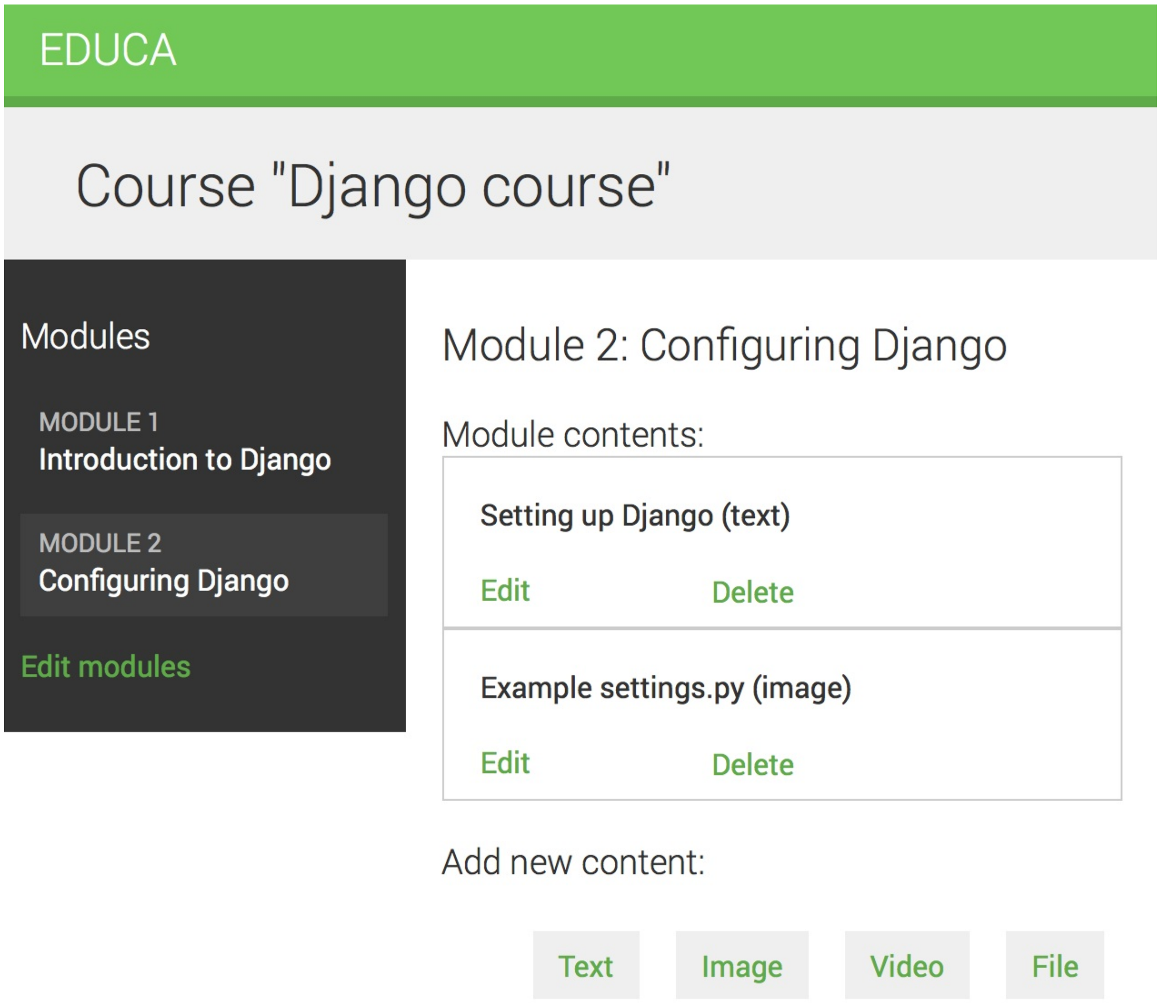
```
<a href="{% url "course_module_update" course.id %}">Edit modules</a>
{% if course.modules.count > 0 %}
  <a href="{% url "module_content_list" course.modules.first.id %}">
  Manage contents</a>
{% endif %}
```

The new link allows users to access the contents of the first module of the course, if any.

Open `http://127.0.0.1:8000/course/mine/` and click the Manage contents link for a course that contains at least one module. You will see a page like the following one:



When you click on a module in the left sidebar, its contents are displayed in the main area. The template also includes links to add a new text, video, image, or file content for the module being displayed. Add a couple of different types of content to the module and take a look at the result. The contents will appear after Module contents like in the following example:


EDUCA
Course "Django course"
Modules
MODULE 1
Introduction to Django
MODULE 2
Configuring Django
Edit modules
Module 2: Configuring Django
Module contents:
Setting up Django (text)
Edit
Delete
Example settings.py (image)
Edit
Delete
Add new content:
Text
Image
Video
File

Reordering modules and contents

We need to provide a simple way to reorder course modules and their contents. We will use a JavaScript drag-n-drop widget to let our users reorder the modules of a course by dragging them. When users finish dragging a module, we will launch an asynchronous request (AJAX) to store the new module order.

Using mixins from django-braces

`django-braces` is a third-party module that contains a collection of generic mixins for Django. These mixins provide additional features for class-based views. You can see a list of all mixins provided by `django-braces` at `https://django-braces.readthedocs.io/`.

We will use the following mixins of `django-braces`:

- `CsrfExemptMixin`: To avoid checking the CSRF token in the `POST` requests. We need this to perform AJAX `POST` requests without having to generate a `csrf_token`.
- `JsonRequestResponseMixin`: Parses the request data as JSON and also serializes the response as JSON and returns an HTTP response with the `application/json` content type.

Install `django-braces` via `pip` using the following command:

```
pip install django-braces==1.13.0
```

We need a view that receives the new order of modules' ID encoded in JSON. Edit the `views.py` file of the `courses` application and add the following code to it:

```
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'})
```

This is the `ModuleOrderView` view.

We can build a similar view to order a module's contents. Add the following code to the `views.py` file:

```
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'})
```

Now, edit the `urls.py` file of the `courses` application and add the following URL patterns to it:

```
path('module/order/',
     views.ModuleOrderView.as_view(),
     name='module_order'),

path('content/order/',
     views.ContentOrderView.as_view(),
     name='content_order'),
```

Finally, we need to implement the drag-n-drop functionality in the template. We will use the jQuery UI library for this. jQuery UI is built on top of jQuery and it provides a set of interface interactions, effects, and widgets. We will use its `sortable` element. First, we need to load jQuery UI in the base template. Open the `base.html` file located in the `templates/` directory of the `courses` application, and add jQuery UI below the script to load jQuery as follows:

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-
ui.min.js"></script>
```

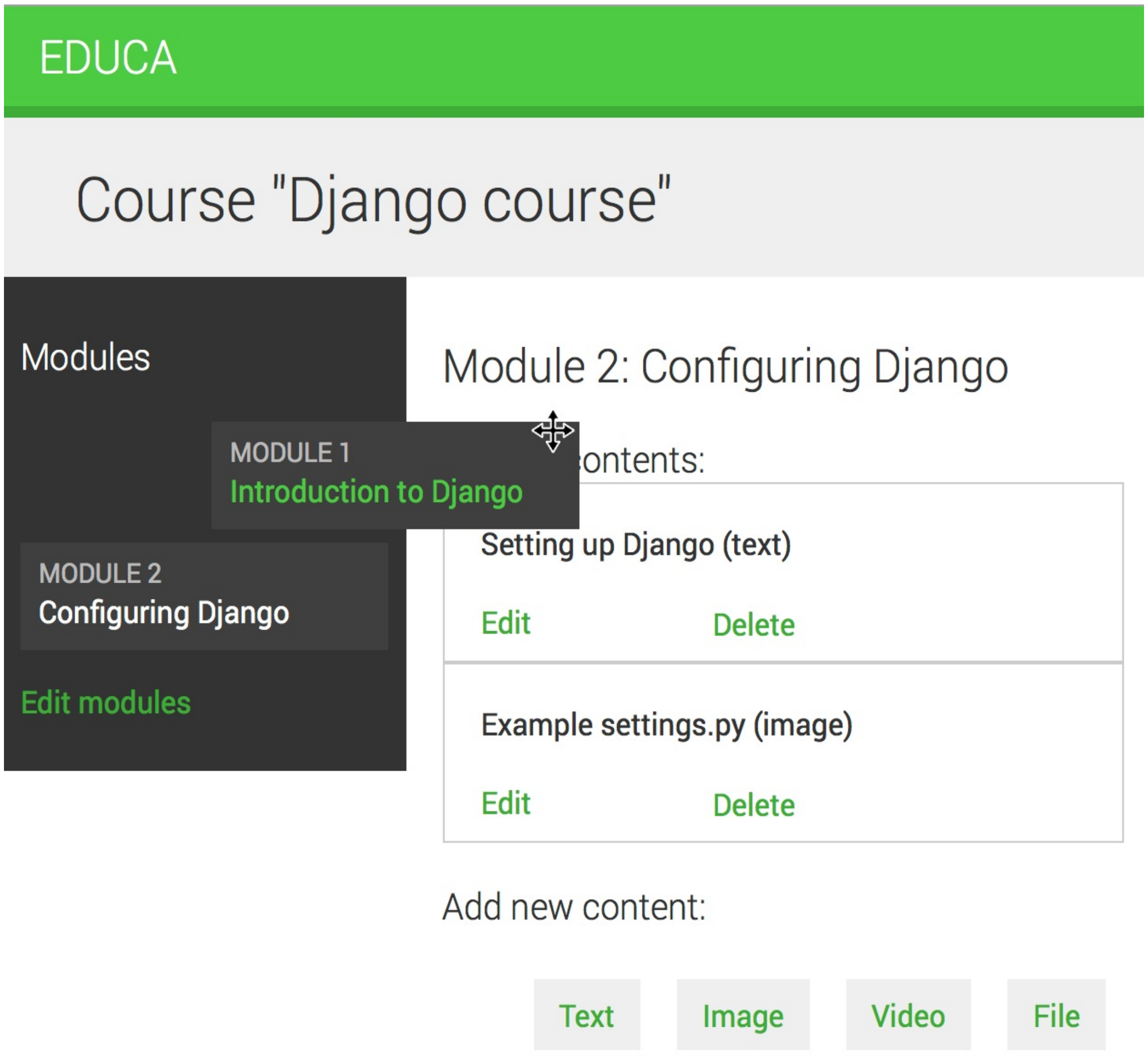
We load the jQuery UI library just after the jQuery framework. Now, edit the `courses/manage/module/content_list.html` template and add the following code to it at the bottom of the template:

```
{% block domready %}
$('#modules').sortable({
    stop: function(event, ui) {
        modules_order = {};
        $('#modules').children().each(function(){
            // update the order field
            $(this).find('.order').text($(this).index() + 1);
            // associate the module's id with its order
            modules_order[$(this).data('id')] = $(this).index();
        });
        $.ajax({
            type: 'POST',
            url: '{% url "module_order" %}',
            contentType: 'application/json; charset=utf-8',
            dataType: 'json',
               data: JSON.stringify(modules_order)
           });
    }
});

$('#module-contents').sortable({
    stop: function(event, ui) {
        contents_order = {};
        $('#module-contents').children().each(function(){
            // associate the module's id with its order
            contents_order[$(this).data('id')] = $(this).index();
        });

        $.ajax({
            type: 'POST',
            url: '{% url "content_order" %}',
            contentType: 'application/json; charset=utf-8',
            dataType: 'json',
            data: JSON.stringify(contents_order),
        });
    }
});
{% endblock %}
```

This JavaScript code is in the `{% block domready %}` block and therefore it will be included in the `$(document).ready()` event of jQuery that we defined in the `base.html` template. This guarantees that our JavaScript code is executed once the page has been loaded. We define a `sortable` element for the modules list in the sidebar and a different one for the module's content list. Both work in a similar manner. In this code, we perform the following tasks:

1. First, we define a `sortable` element for the `modules` HTML element. Remember that we use `#modules`, since jQuery uses CSS notation for selectors.
2. We specify a function for the `stop` event. This event is triggered every time the user finishes sorting an element.
3. We create an empty `modules_order` dictionary. The keys for this dictionary will be the modules' ID, and the values will be the assigned order for each module.
4. We iterate over the `#module` children elements. We recalculate the displayed order for each module and get its `data-id` attribute, which contains the module's ID. We add the ID as the key of the `modules_order` dictionary and the new index of the module as the value.
5. We launch an AJAX `POST` request to the `content_order` URL, including the serialized JSON data of `modules_order` in the request. The corresponding `ModuleOrderView` takes care of updating the modules' order.

The `sortable` element to order contents is quite similar to this one. Go back to your browser and reload the page. Now, you will be able to click and drag both modules and contents to reorder them like the following example:



Great! Now you can reorder both course modules and module contents.

Summary

In this chapter, you learned how to create a versatile CMS. You used model inheritance and created a custom model field. You also worked with class-based views and mixins. You created formsets and a system to manage diverse types of content.

In the next chapter, you will create a student registration system. You will also render different kinds of content, and you will learn how to work with Django's cache framework.

Rendering and Caching Content

In the previous chapter, you used model inheritance and generic relationships to create flexible course content models. You also built a course management system using class-based views, formsets, and AJAX ordering for contents. In this chapter, you will:

- Create public views for displaying course information
- Build a student registration system
- Manage student enrollment in courses
- Render diverse course contents
- Cache content using the cache framework

We will start by creating a course catalog for students to browse existing courses and be able to enroll in them.

Displaying courses

For our course catalog, we have to build the following functionality:

- List all available courses, optionally filtered by subject
- Display a single course overview

Edit the `views.py` file of the `courses` application and add the following code:

```
from django.db.models import Count
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})
```

This is the `CourseListView` view. It inherits from `TemplateResponseMixin` and `View`. In this view, we perform the following tasks:

1. We retrieve all subjects, including the total number of courses for each of them. We use the ORM's `annotate()`

method with the `Count()` aggregation function to include the total number of courses for each subject.
2. We retrieve all available courses, including the total number of modules contained in each course.
3. If a subject slug URL parameter is given, we retrieve the corresponding subject object and we limit the query to the courses that belong to the given subject.
4. We use the `render_to_response()` method provided by `TemplateResponseMixin` to render the objects to a template and return an HTTP response.

Let's create a detail view for displaying a single course overview. Add the following code to the `views.py` file:

```
from django.views.generic.detail import DetailView

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'
```

This view inherits from the generic `DetailView` provided by Django. We specify the `model` and `template_name` attributes. Django's `DetailView` expects a primary key (`pk`) or slug URL parameter to retrieve a single object for the given model. Then, it renders the template specified in `template_name`, including the object in the context as object.

Edit the main `urls.py` file of the `educa` project and add the following URL pattern to it:

```
from courses.views import CourseListView

urlpatterns = [
    # ...
    path('', CourseListView.as_view(), name='course_list'),
]
```

We add the `course_list` URL pattern to the main `urls.py` file of the project because we want to display the list of courses in the URL `http://127.0.0.1:8000/` and all other URLs for the `courses` application have the `/course/` prefix.

Edit the `urls.py` file of the `courses` application and add the following URL patterns:

```
path('subject/<slug:subject>)/',
     views.CourseListView.as_view(),
     name='course_list_subject'),

path('<slug:slug>/',
     views.CourseDetailView.as_view(),
     name='course_detail'),
```

We define the following URL patterns:

- `course_list_subject`: For displaying all courses for a subject
- `course_detail`: For displaying a single course overview

Let's build templates for the `CourseListView` and `CourseDetailView` views. Create the following file structure inside the `templates/courses/` directory of the courses application:

```
course/
    list.html
    detail.html
```

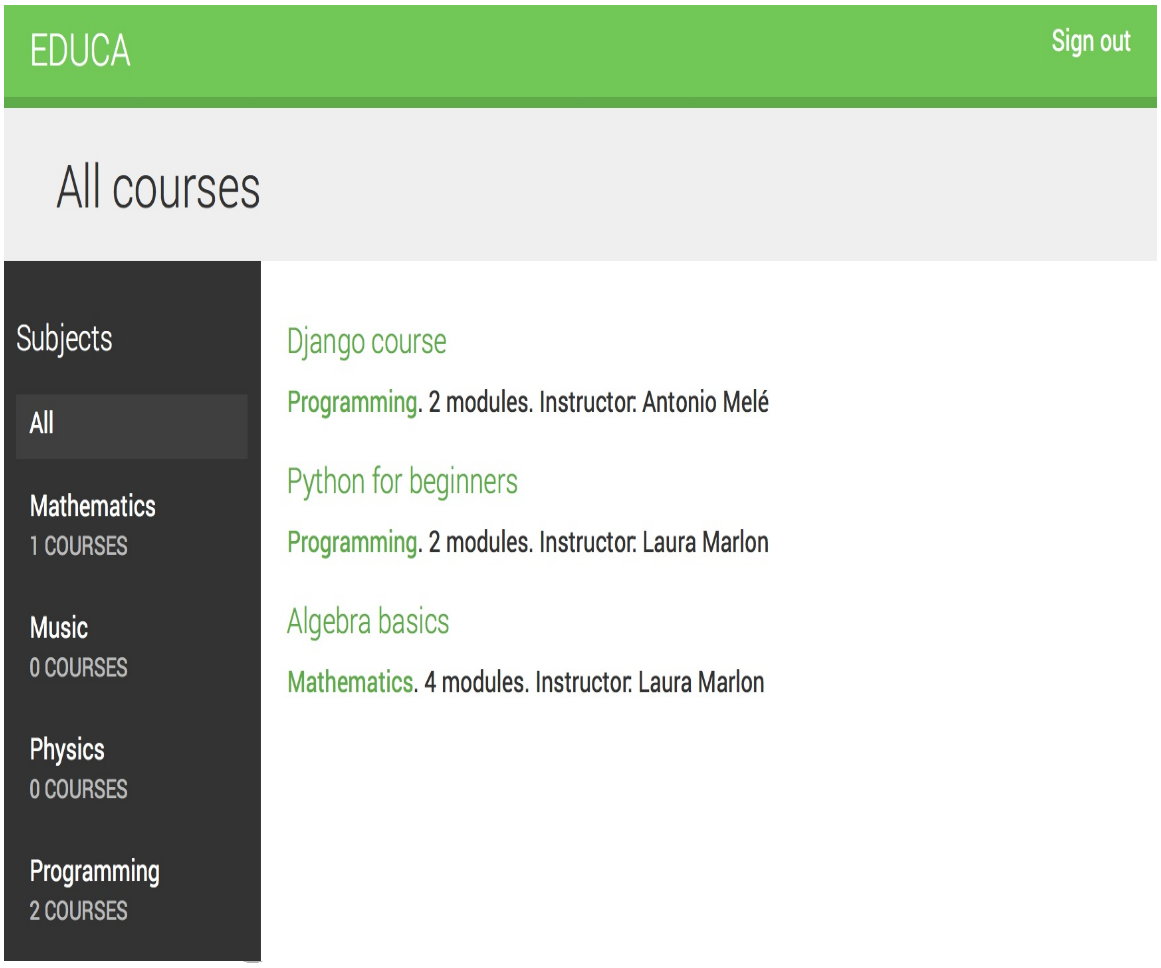
Edit the `courses/course/list.html` template and write the following code:

```
{% 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 }} courses</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 %}
```

This is the template for listing the available courses. We create an HTML list to display all `Subject` objects and build a link to the `course_list_subject` URL for each of them. We add a `selected` HTML class to highlight the current subject, if any. We iterate over every `Course` object, displaying the total number of modules and the instructor name.

Run the development server and open `http://127.0.0.1:8000/` in your browser. You should see a page similar to the following one:


EDUCA

Sign out

All courses

Subjects

All

Mathematics
1 COURSES

Music
0 COURSES

Physics
0 COURSES

Programming
2 COURSES

Django course

Programming. 2 modules. Instructor: Antonio Melé

Python for beginners

Programming. 2 modules. Instructor: Laura Marlon

Algebra basics

Mathematics. 4 modules. Instructor: Laura Marlon

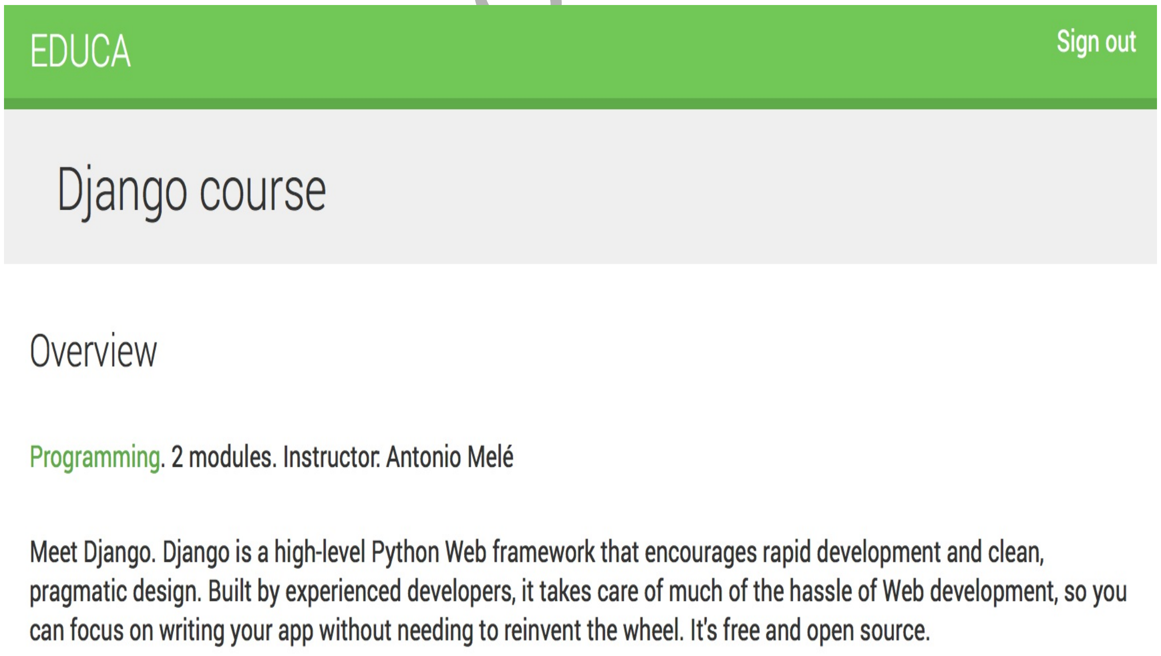
The left sidebar contains all subjects, including the total number of courses for each of them. You can click any subject to filter the courses displayed.

Edit the `courses/course/detail.html` template and add the following code to it:

```
{% extends "base.html" %}

{% block title %}
```

```
    {{ object.title }}
{% endblock %}

{% block content %}
    {% with subject=course.subject %}
        <h1>
            {{ object.title }}
        </h1>
        <div class="module">
            <h2>Overview</h2>
            <p>
                <a href="{% url "course_list_subject" subject.slug %}">
                {{ subject.title }}</a>.
                {{ course.modules.count }} modules.
                Instructor: {{ course.owner.get_full_name }}
            </p>
            {{ object.overview|linebreaks }}
        </div>
    {% endwith %}
{% endblock %}
```

In this template, we display the overview and details for a single course. Open `http://127.0.0.1:8000/` in your browser and click one of the courses. You should see a page with the following structure:



Django course

Overview

Programming. 2 modules. Instructor: Antonio Melé

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

We have created a public area for displaying courses. Next, we need to allow users to register as students and enroll in courses.

Adding student registration

Create a new application using the following command:

```
python manage.py startapp students
```

Edit the `settings.py` file of the `educa` project and add the new application to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    # ...
    'students.apps.StudentsConfig',
]
```

Creating a student registration view

Edit the `views.py` file of the `students` application and write the following code:

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login

class StudentRegistrationView(CreateView):
    template_name = 'students/student/registration.html'
    form_class = UserCreationForm
    success_url = reverse_lazy('student_course_list')

    def form_valid(self, form):
        result = super(StudentRegistrationView,
                       self).form_valid(form)
        cd = form.cleaned_data
        user = authenticate(username=cd['username'],
                            password=cd['password1'])
        login(self.request, user)
        return result
```

This is the view that allows students to register on our site. We use the generic `CreateView`, which provides the functionality for creating model objects. This view requires the following attributes:

- `template_name`: The path of the template to render this view.
- `form_class`: The form for creating objects, which has to be `ModelForm`. We use Django's `UserCreationForm` as the registration form to create `User` objects.

- `success_url`: The URL to redirect the user to when the form is successfully submitted. We reverse the `student_course_list` URL, which we are going to create in the *Accessing the course contents* section for listing the courses students are enrolled in.

The `form_valid()` method is executed when valid form data has been posted. It has to return an HTTP response. We override this method to log the user in after successfully signing up.

Create a new file inside the `students` application directory and name it `urls.py`. Add the following code to it:

```
from django.urls import path
from . import views

urlpatterns = [
    path('register/',
         views.StudentRegistrationView.as_view(),
         name='student_registration'),
]
```

Then, edit the main `urls.py` of the `educa` project and include the URLs for the `students` application by adding the following pattern to your URL configuration:

```
urlpatterns = [
    # ...
    path('students/', include('students.urls')),
]
```

Create the following file structure inside the `students` application directory:

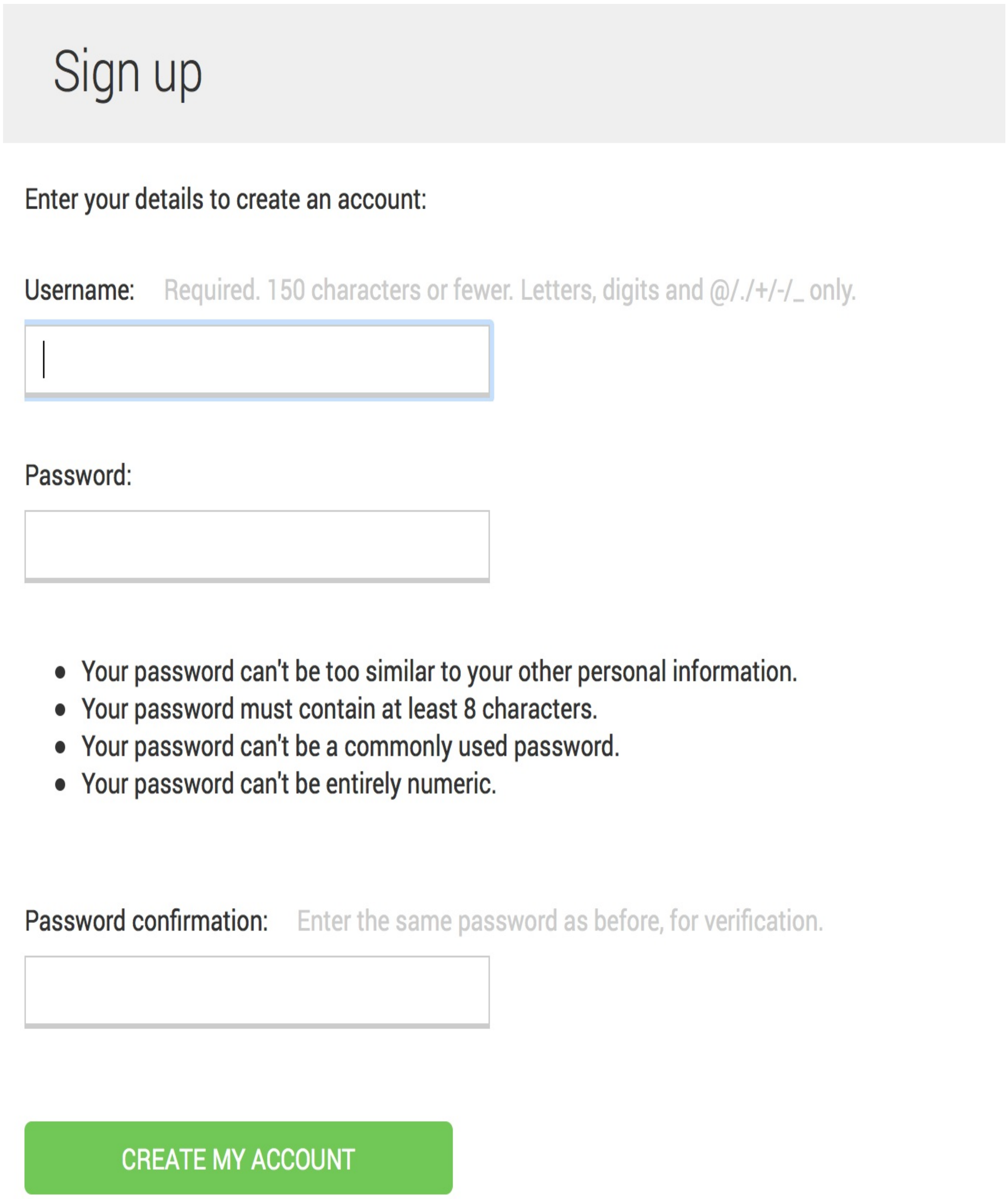
```
templates/
    students/
        student/
```

```
            registration.html
```

Edit the `students/student/registration.html` template and add the following code to it:

```
{% extends "base.html" %}

{% block title %}
    Sign up
{% endblock %}

{% block content %}
    <h1>
        Sign up
    </h1>
    <div class="module">
        <p>Enter your details to create an account:</p>
        <form action="" method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <p><input type="submit" value="Create my account"></p>
        </form>
    </div>
{% endblock %}
```

Run the development server and open `http://127.0.0.1:8000/students/register/` in your browser. You should see the registration form like this:



Sign up

Enter your details to create an account:

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Note that the `student_course_list` URL specified in the `success_url` attribute of the `StudentRegistrationView` view doesn't exist yet. If you submit the form, Django won't find the URL to redirect you after a successful registration. We will create this URL in the *Accessing the course contents* section.

Enrolling in courses

After users create an account, they should be able to enroll in `courses`. In order to store enrollments, we need to create a many-to-many relationship between the `Course` and `User` models.

Edit the `models.py` file of the `courses` application and add the following field to the `Course` model:

```
students = models.ManyToManyField(User,
                                  related_name='courses_joined',
                                  blank=True)
```

From the shell, execute the following command to create a migration for this change:

```
python manage.py makemigrations
```

You will see an output similar to this:

```
Migrations for 'courses':
  courses/migrations/0004_course_students.py
    - Add field students to course
```

Then, execute the next command to apply pending migrations:

```
python manage.py migrate
```

You should see output that ends with the following line:

```
Applying courses.0004_course_students... OK
```

We can now associate students with the courses in which they are enrolled. Let's create the functionality for students to enroll in courses.

Create a new file inside the `students` application directory and name it `forms.py`. Add the following code to it:

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
    course = forms.ModelChoiceField(queryset=Course.objects.all(),
                                    widget=forms.HiddenInput)
```

We are going to use this form for students to enroll in courses. The `course` field is for the course in which the user gets enrolled. Therefore, it's a `ModelChoiceField`. We use a `HiddenInput` widget because we are not going to show this field to the user. We are going to use this form in the `CourseDetailView` view to display a button to enroll.

Edit the `views.py` file of the `students` application and add the following code:

```
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(StudentEnrollCourseView,
                     self).form_valid(form)

    def get_success_url(self):
        return reverse_lazy('student_course_detail',
                            args=[self.course.id])
```

This is the `StudentEnrollCourseView`. view. It handles students enrolling in `courses`. The view inherits from the `LoginRequiredMixin` mixin so that only logged in users can access the view. It also inherits from Django's `FormView` view since we handle a form submission. We use the `CourseEnrollForm` form for the `form_class` attribute and also define a `course` attribute for storing the given `Course` object. When the form is valid, we add the current user to the students enrolled in the course.

The `get_success_url()` method returns the URL the user will be redirected to if the form was successfully submitted. This method is equivalent to the `success_url` attribute. We reverse the `student_course_detail` URL, which we will create in the next *Accessing the course contents* section in order to display the course contents.

Edit the `urls.py` file of the `students` application and add the following URL pattern to it:

```
path('enroll-course/',
     views.StudentEnrollCourseView.as_view(),
     name='student_enroll_course'),
```

Let's add the enroll button form to the course overview page. Edit the `views.py` file of the `courses` application and modify `CourseDetailView` to make it look as follows:

```
from students.forms import CourseEnrollForm

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'

    def get_context_data(self, **kwargs):
        context = super(CourseDetailView,
                        self).get_context_data(**kwargs)
        context['enroll_form'] = CourseEnrollForm(
                                   initial={'course':self.object})
        return context
```

We use the `get_context_data()` method to include the enrollment form in the context for rendering the templates. We initialize the hidden course field of the form with the current `Course` object, so that it can be submitted directly.

Edit the `courses/course/detail.html` template and find the following line:

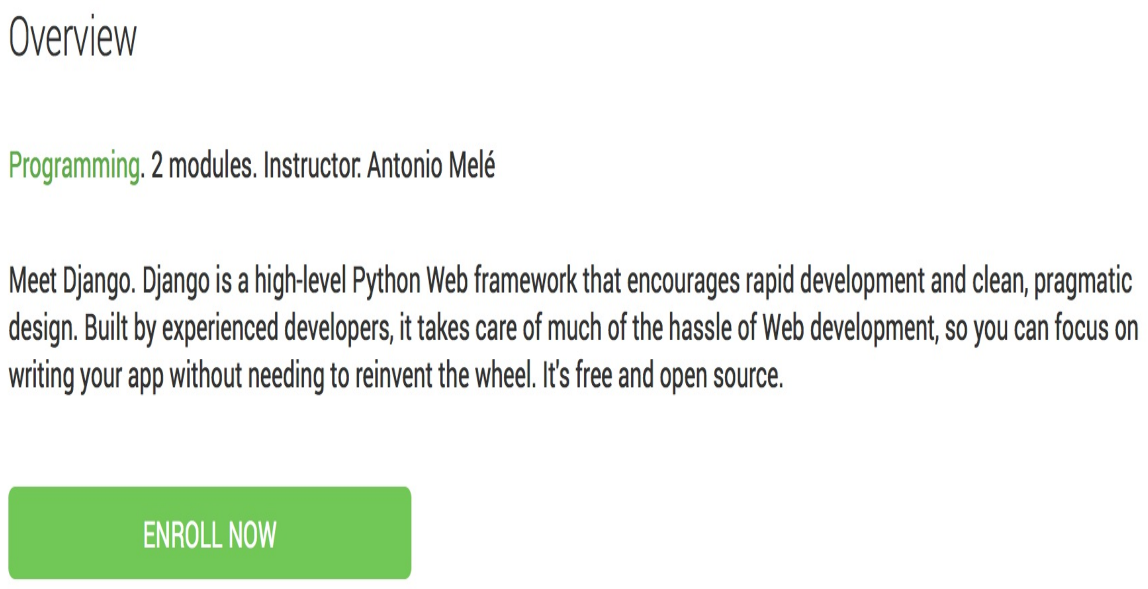
```
{{ object.overview|linebreaks }}
```

Replace it with the following code:

```
{{ object.overview|linebreaks }}
{% if request.user.is_authenticated %}
    <form action="{% url "student_enroll_course" %}" method="post">
        {{ enroll_form }}
        {% csrf_token %}
        <input type="submit" class="button" value="Enroll now">
    </form>
{% else %}
    <a href="{% url "student_registration" %}" class="button">
        Register to enroll
    </a>
{% endif %}
```

This is the button for enrolling in courses. If the user is authenticated, we display the enrollment button, including the hidden form that points to the `student_enroll_course` URL. If the user is not authenticated, we display a link to register in the platform.

Make sure the development server is running, open `http://127.0.0.1:8000/` in your browser and click a course. If you are logged in, you should see an ENROLL NOW button placed below the course overview, as follows:



Overview

Programming. 2 modules. Instructor. Antonio Melé

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

If you are not logged in, you will see a REGISTER TO ENROLL button instead.

Accessing the course contents

We need a view for displaying the courses the students are enrolled in, and a view for accessing the actual course contents. Edit the `views.py` file of the `students` application and add the following code to it:

```
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(StudentCourseListView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])
```

This is the view for students to list the courses they are enrolled in. It inherits from `LoginRequiredMixin` to make sure that only logged in users can access the view. It also inherits from the generic `ListView` for displaying a list of `Course` objects. We override the `get_queryset()` method for retrieving only the courses the user is enrolled in; we filter the QuerySet by the student's `ManyToManyField` field for doing so.

Then, add the following code to the `views.py` file:

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
    model = Course
    template_name = 'students/course/detail.html'

    def get_queryset(self):
        qs = super(StudentCourseDetailView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])

    def get_context_data(self, **kwargs):
        context = super(StudentCourseDetailView,
                        self).get_context_data(**kwargs)
        # get course object
        course = self.get_object()
        if 'module_id' in self.kwargs:
            # get current module
            context['module'] = course.modules.get(
                                    id=self.kwargs['module_id'])
        else:
            # get first module
            context['module'] = course.modules.all()[0]
        return context
```

This is `StudentCourseDetailView`. We override the `get_queryset()` method to limit the base QuerySet to courses in which the user is enrolled. We also override the `get_context_data()` method to set a course module in the context if the `module_id` URL parameter is given. Otherwise, we set the first module of the course. This way, students will be able to navigate through modules inside a course.

Edit the `urls.py` file of the `students` application and add the following URL patterns to it:

```
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'),
```

Create the following file structure inside the `templates/students/` directory of the `students` application:

```
course/
    detail.html
    list.html
```

Edit the `students/course/list.html` template and add the following code to it:

```
{% 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 in a course.
            </p>
        {% endfor %}
    </div>
{% endblock %}
```

This template displays the courses the user is enrolled in. Remember that when a new student successfully registers with the platform, they will be redirected to the `student_course_list` URL. Let's also redirect students to this URL when they log in to the platform.

Edit the `settings.py` file of the `educa` project and add the following code to it:

```
from django.urls import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')
```

This is the setting used by the `auth` module to redirect the user to after a successful login if no next parameter is present in the request. After successful login, students will be redirected to the `student_course_list` URL to view the courses that they are enrolled in.

Edit the `students/course/detail.html` template and add the following code to it:

```
{% 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 %}
```

This is the template for enrolled students to access the contents of a course. First, we build an HTML list including all course modules and highlighting the current module. Then, we iterate over the current module contents and access each content item to display it using `{{ item.render }}`. We are going to add the `render()` method to the content models next. This method will take care of rendering the content properly.

Rendering different types of content

We need to provide a way to render each type of content. Edit the `models.py` file of the `courses` application and add the following `render()` method to the `ItemBase` model:

```
from django.template.loader import render_to_string
from django.utils.safestring import mark_safe

class ItemBase(models.Model):
    # ...

    def render(self):
        return render_to_string('courses/content/{}.html'.format(
                    self._meta.model_name), {'item': self})
```

This method uses the `render_to_string()` function for rendering a template and returning the rendered content as a string. Each kind of content is rendered using a template named after the content model. We use `self._meta.model_name` to generate the appropriate template name for each content model dynamically. The `render()` method provides a common interface for rendering diverse content.

Create the following file structure inside the `templates/courses/` directory of the courses application:

```
content/
    text.html
    file.html
    image.html
    video.html
```

Edit the `courses/content/text.html` template and write this code:

```
{{ item.content|linebreaks|safe }}
```

Edit the `courses/content/file.html` template and add the following:

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

Edit the `courses/content/image.html` template and write:

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

For files uploaded with `ImageField` and `FileField` to work, we need to set up our project to serve media files with the development server. Edit the `settings.py` file of your project and add the following code to it:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

Remember that `MEDIA_URL` is the base URL to serve uploaded media files and `MEDIA_ROOT` is the local path where the files are located.

Edit the main `urls.py` file of your project and add the following imports:

```
from django.conf import settings
from django.conf.urls.static import static
```

Then, write the following lines at the end of the file:

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Your project is now ready to upload and serve media files. The Django development server will be in charge of serving the media files during development (that is, when the `DEBUG` setting is set to `True`).

We also have to create a template for rendering `Video` objects. We will use `django-embed-video` for embedding video content. `django-embed-video` is a third-party Django application that allows you to embed videos in your templates, from sources such as YouTube or Vimeo, by simply providing the video's public URL.

Install the package with the following command:

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

Edit the `settings.py` file of your project and add the app to the `INSTALLED_APPS`, setting as follows:

```
INSTALLED_APPS = [
    # ...
    'embed_video',
]
```

You can find `django-embed-video` application's documentation at `https://django-embed-video.readthedocs.io/en/latest/`.

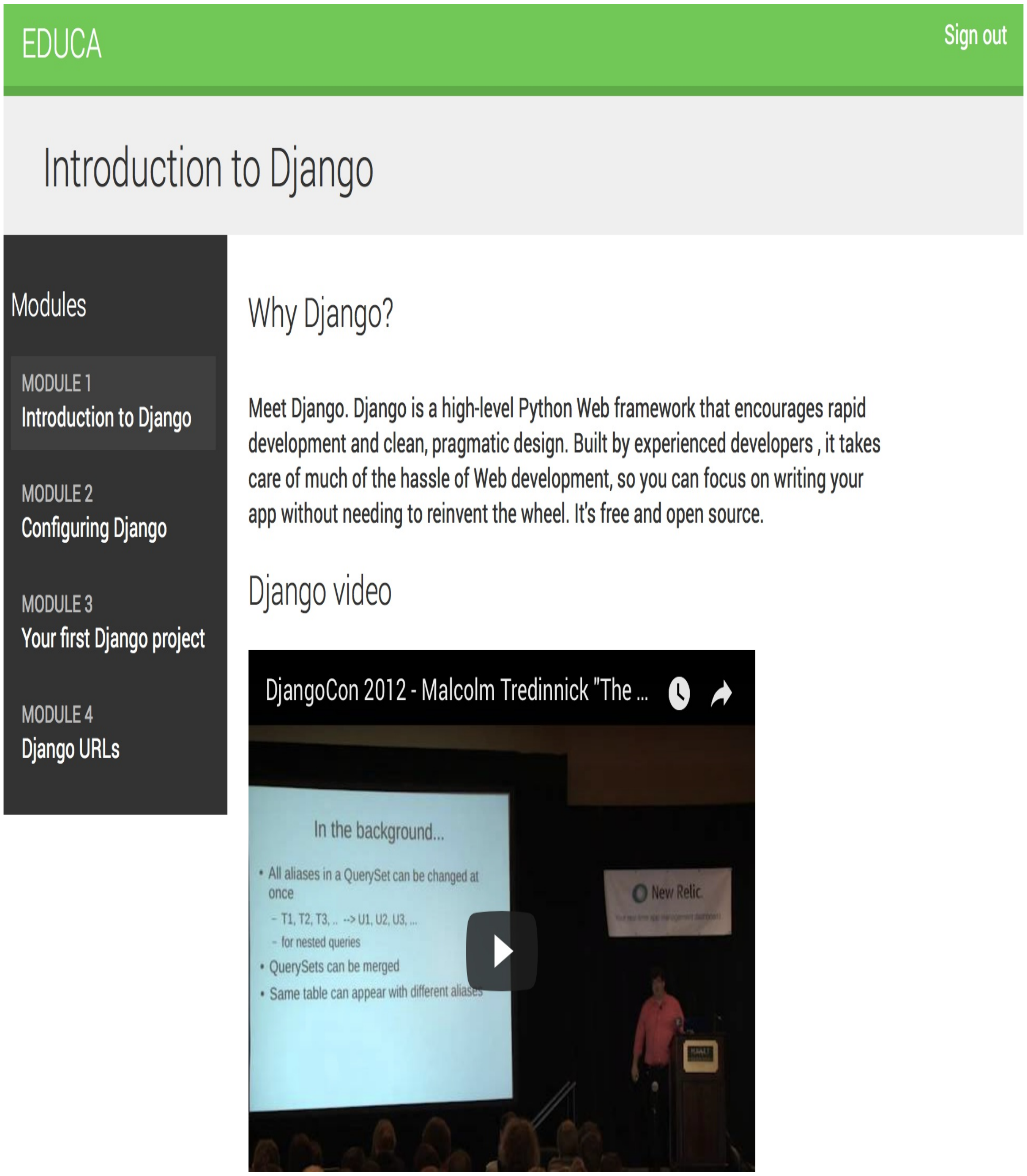
Edit the `courses/content/video.html` template and write the following code:

```
{% load embed_video_tags %}
{% video item.url "small" %}
```

Now run the development server and access `http://127.0.0.1:8000/course/mine/` in your browser.

Access the site with a user that belongs to the `Instructors` group, and add multiple contents to a course. To include video content, you can just copy any YouTube URL, such as `https://www.youtube.com/watch?v=bgV39DlmZ2U`, and include it in the `url` field of the form.

After adding contents to the course open `http://127.0.0.1:8000/`, click the course and click on the ENROLL NOW button. You should be enrolled in the course and redirected to the `student_course_detail` URL. The following screenshot shows a sample course content:



EDUCA

Sign out

Introduction to Django

Modules

MODULE 1
Introduction to Django

MODULE 2
Configuring Django

MODULE 3
Your first Django project

MODULE 4
Django URLs

Why Django?

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers , it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Django video

DjangoCon 2012 - Malcolm Tredinnick "The ...

In the background...
All aliases in a QuerySet can be changed at once
T1, T2, T3, .. --> U1, U2, U3, ...
for nested queries
QuerySets can be merged
Same table can appear with different aliases
New Relic

Great! You have created a common interface for rendering different types of course contents.

Using the cache framework

HTTP requests to your web application usually entail database access, data processing, and template rendering. This is much more expensive in terms of processing than serving a static website.

The overhead in some requests can be significant when your site starts getting more and more traffic. This is where caching becomes precious. By caching queries, calculation results, or rendered content in an HTTP request, you will avoid cost-expensive operations in the following requests. This translates into shorter response times and less processing on the server side.

Django includes a robust cache system that allows you to cache data with different levels of granularity. You can cache a single query, the output of a specific view, parts of rendered template content, or your entire site. Items are stored in the cache system for a default time. You can specify the default timeout for cached data.

This is how you will usually use the cache framework when your application gets an HTTP request:

1. Try to find the requested data in the cache
2. If found, return the cached data
3. If not found, perform the following steps:

    1. Perform the query or processing required to obtain the data
    2. Save the generated data in the cache
    3. Return the data

You can read detailed information about Django's cache system at `https://docs.djangoproject.com/en/2.0/topics/cache/`.

Available cache backends

Django comes with several cache backends. These are the following:

- `backends.memcached.MemcachedCache` or `backends.memcached.PyLibMCCache`: A Memcached backend. Memcached is a fast and efficient memory-based cache server. The backend to use depends on the Memcached Python bindings you choose.
- `backends.db.DatabaseCache`: Use the database as cache system.
- `backends.filebased.FileBasedCache`: Use the file storage system. Serializes and stores each cache value as a separate file.
- `backends.locmem.LocMemCache`: Local memory cache backend. This the default cache backend.
- `backends.dummy.DummyCache`: A dummy cache backend intended only for development. It implements the cache interface without actually caching anything. This cache is per-process and thread-safe.

For optimal performance, use a memory-based cache backend such as the Memcached backend.

Installing Memcached

We are going to use the Memcached backend. Memcached runs in memory and it is allotted a specified amount of RAM. When the allotted RAM is full, Memcached starts removing the oldest data to store new data.

Download Memcached from `https://memcached.org/downloads`. If you are using Linux, you can install Memcached using the following command:

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

If you are using macOS X, you can install Memcached with the Homebrew package
manager using the command `brew install memcached`. You can download Homebrew from `https://brew.sh/`.

After installing Memcached, open a shell and start it using the following command:

```
memcached -l 127.0.0.1:11211
```

Memcached will run on port `11211` by default. However, you can specify a custom host and port by using the `-l` option. You can find more information about Memcached at `https://memcached.org`.

After installing Memcached, you have to install its Python bindings. You can do it with the following command:

```
pip install python-memcached==1.59
```

Cache settings

Django provides the following cache settings:

- `CACHES`: A dictionary containing all available caches for the project.
- `CACHE_MIDDLEWARE_ALIAS`: The cache alias to use for storage.
- `CACHE_MIDDLEWARE_KEY_PREFIX`: The prefix to use for cache keys. Set a prefix to avoid key collisions if you share the same cache between several sites.
- `CACHE_MIDDLEWARE_SECONDS`: The default number of seconds to cache pages.

The caching system for the project can be configured using the `CACHES` setting. This setting is a dictionary that allows you to specify the configuration for multiple caches. Each cache included in the `CACHES` dictionary can specify the following data:

- `BACKEND`: The cache backend to use.
- `KEY_FUNCTION`: A string containing a dotted path to a callable that takes a prefix, version, and key as arguments and returns a final cache key.
- `KEY_PREFIX`: A string prefix for all cache keys, to avoid collisions.
- `LOCATION`: The location of the cache. Depending on the cache

backend, this might be a directory, a host and port, or a name for the in-memory backend.

- `OPTIONS`: Any additional parameters to be passed to the cache backend.
- `TIMEOUT`: The default timeout, in seconds, for storing the cache keys. 300 seconds by default, which is five minutes. If set to `None`, cache keys will not expire.
- `VERSION`: The default version number for the cache keys. Useful for cache versioning.

Adding Memcached to your project

Let's configure the cache for our project. Edit the `settings.py` file of the `educa` project and add the following code to it:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

We are using the `MemcachedCache` backend. We specify its location using the `address:port` notation. If you have multiple Memcached instances, you can use a list for `LOCATION`.

Monitoring Memcached

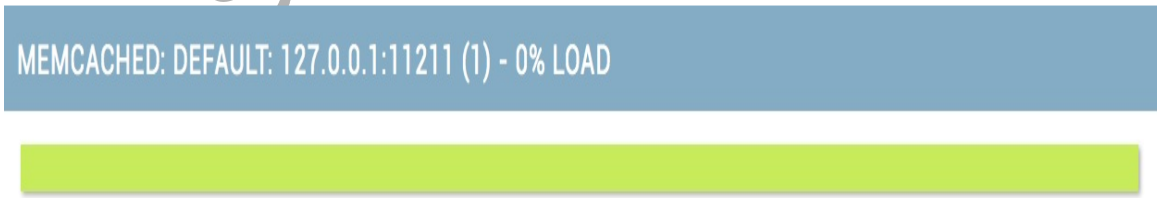
In order to monitor Memcached, we will use a third-party package called `django-memcache-status`. This app displays statistics for your Memcached instances in the administration site. Install it with the following command:

```
pip install django-memcache-status==1.3
```

Edit the `settings.py` file and add `'memcache_status'` to the `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # ...
    'memcache_status',
]
```

Make sure Memcached is running, start the development server in another shell window, and open `http://127.0.0.1:8000/admin/` in your browser. Log in to the administration site using a superuser. You should see the following block:



This graph shows the cache usage. The green color represents free cache while red indicates used space. If you click the title of the box, it shows detailed statistics of your Memcached instance.

We have set up Memcached for our project and are able to monitor it. Let's start caching data!

Cache levels

Django provides the following levels of caching listed here by ascending order of granularity:

- **Low-level cache API**: Provides the highest granularity. Allows you to cache specific queries or calculations.
- **Per-view cache**: Provides caching for individual views.
- **Template cache**: Allows you to cache template fragments.
- **Per-site cache**: The highest-level cache. It caches your entire site.

Think about your cache strategy before implementing caching. Focus first on expensive queries or calculations, which are not calculated on a per-user basis.

Using the low-level cache API

The low-level cache API allows you to store objects in the cache with any granularity. It is located at `django.core.cache`. You can import it like this:

```
from django.core.cache import cache
```

This uses the default cache. It's equivalent to `caches['default']`. Accessing a specific cache is also possible via its alias:

```
from django.core.cache import caches
my_cache = caches['alias']
```

Let's take a look at how the cache API works. Open the shell with the command `python manage.py shell` and execute the following code:

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

We access the default cache backend and use `set(key, value, timeout)` to store a key named `'musician'` with a value that is the string `'Django Reinhardt'` for 20 seconds. If we don't specify a timeout, Django uses the default timeout specified for the cache backend in the `CACHES` setting. Now, execute the following code:

```
>>> cache.get('musician')
'Django Reinhardt'
```

We retrieve the key from the cache. Wait for 20 seconds and execute the same code:

```
>>> cache.get('musician')
```

No value is returned this time. The `'musician'` cache key expired and the `get()` method returns `None` because the key is not in the cache anymore.

> *Always avoid storing a `None` value in a cache key because you won't be able to distinguish between the actual value and a cache miss.*

Let's cache a QuerySet with the following code:

```
>>> from courses.models import Subject
>>> subjects = Subject.objects.all()
>>> cache.set('all_subjects', subjects)
```

We perform a QuerySet on the `Subject` model and store the returned objects in the `'all_subjects'` key. Let's retrieve the cached data:

```
>>> cache.get('all_subjects')
<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,
<Subject: Programming>]>
```

We are going to cache some queries in our views. Edit the `views.py` file of the `courses` application and add the following import:

```
from django.core.cache import cache
```

In the `get()` method of the `CourseListView`, replace the following line:

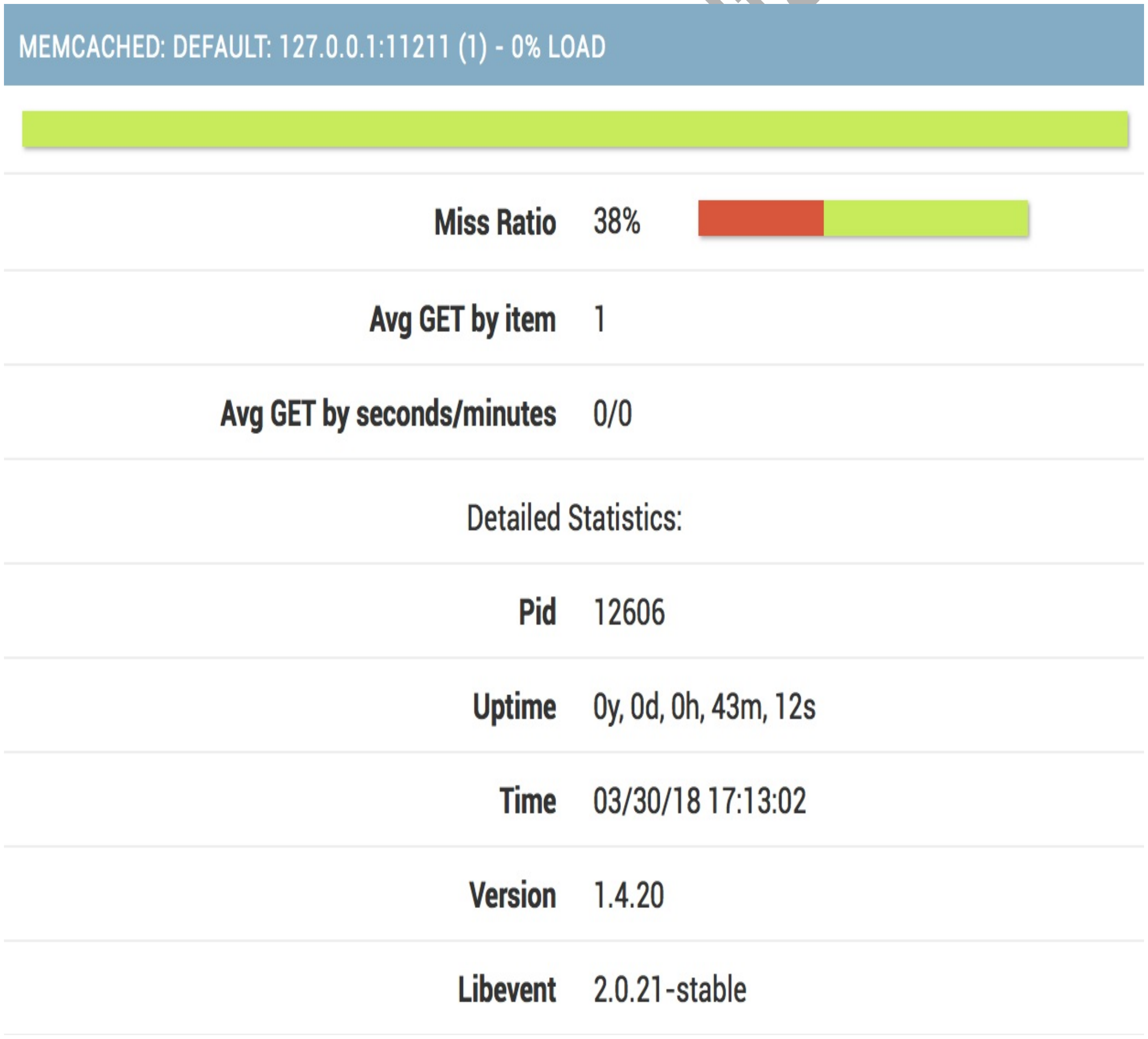
```
subjects = Subject.objects.annotate(
               total_courses=Count('courses'))
```

Replace it with the following ones:

```
subjects = cache.get('all_subjects')
if not subjects:
    subjects = Subject.objects.annotate(
                     total_courses=Count('courses'))
    cache.set('all_subjects', subjects)
```

In this code, we try to get the `all_students` key from the cache using `cache.get()`. This returns `None` if the given key is not found. If no key is found (not cached yet or cached but timed out), we perform the query to retrieve all `Subject` objects and their number of courses, and we cache the result using `cache.set()`.

Run the development server and open `http://127.0.0.1:8000/` in your browser. When the view is executed, the cache key is not found and the QuerySet is executed. Open `http://127.0.0.1:8000/admin/` in your browser and expand the Memcached statistics. You should see usage data for the cache similar to the following screen:



MEMCACHED: DEFAULT: 127.0.0.1:11211 (1) - 0% LOAD

Miss Ratio	38%
Avg GET by item	1
Avg GET by seconds/minutes	0/0

Detailed Statistics:

Pid	12606
Uptime	0y, 0d, 0h, 43m, 12s
Time	03/30/18 17:13:02
Version	1.4.20
Libevent	2.0.21-stable

Take a look at Curr Items, which should be 1. This shows that there is one item currently stored in the cache. Get Hits shows how many get commands were successful and Get Misses shows the get requests for keys that are missing. The Miss Ratio is calculated using both of them.

Now, navigate back to `http://127.0.0.1:8000/` using your browser and reload the page several times. If you take a look at the cache statistics now, you will see several more reads (Get Hits and Cmd Get will increase).

Caching based on dynamic data

Many times you will want to cache something that is based on dynamic data. In these cases, you have to build dynamic keys that contain all information required to uniquely identify the cached data. Edit the `views.py` file of the `courses` application and modify the `CourseListView` view to make it look like this:

```
class CourseListView(TemplateResponseMixin, View):
    model = Course
    template_name = 'courses/course/list.html'

    def get(self, request, subject=None):
        subjects = cache.get('all_subjects')
        if not subjects:
            subjects = Subject.objects.annotate(
                           total_courses=Count('courses'))
            cache.set('all_subjects', subjects)
        all_courses = Course.objects.annotate(
                           total_modules=Count('modules'))
        if subject:
            subject = get_object_or_404(Subject, slug=subject)
            key = 'subject_{}_courses'.format(subject.id)
            courses = cache.get(key)
            if not courses:
                courses = all_courses.filter(subject=subject)
                cache.set(key, courses)
        else:
            courses = cache.get('all_courses')
            if not courses:
                courses = all_courses
                cache.set('all_courses', courses)
        return self.render_to_response({'subjects': subjects,
                                        'subject': subject,
                                        'courses': courses})
```

In this case, we also cache both all courses and courses filtered by subject. We use the `all_courses` cache key for storing all courses if no subject is given. If there is a subject, we build the key dynamically with `'subject_{}_courses'.format(subject.id)`.

It is important to note that you cannot use a cached QuerySet to build other QuerySets, since what you cached are actually the results of the QuerySet. So you cannot do the following:

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

Instead, you have to create the base QuerySet `Course.objects.annotate(total_modules=Count('modules'))`, which is not going to be executed until it is forced, and use it to further restrict the QuerySet with `all_courses.filter(subject=subject)` in case the data was not found in the cache.

Caching template fragments

Caching template fragments is a higher-level approach. You need to load the cache template tags in your template using `{% load cache %}`. Then, you will be able to use the `{% cache %}` template tag to cache specific template fragments. You will usually use the template tag as follows:

```
{% cache 300 fragment_name %}
    ...
{% endcache %}
```

The `{% cache %}` tag has two required arguments: the timeout, in seconds, and a name for the fragment. If you need to cache content depending on dynamic data, you can do so by passing additional arguments to the `{% cache %}` template tag to uniquely identify the fragment.

Edit the `/students/course/detail.html` of the `students` application. Add the following code at the top of it, just after the `{% extends %}` tag:

```
{% load cache %}
```

Then, replace the following lines:

```
{% for content in module.contents.all %}
  {% with item=content.item %}
    <h2>{{ item.title }}</h2>
    {{ item.render }}
  {% endwith %}
{% endfor %}
```

Replace them with the following ones:

```
{% cache 600 module_contents module %}
  {% for content in module.contents.all %}
    {% with item=content.item %}
      <h2>{{ item.title }}</h2>
      {{ item.render }}
    {% endwith %}
  {% endfor %}
{% endcache %}
```

We cache this template fragment using the name `module_contents` and passing the current `Module` object to it. Thus, we uniquely identify the fragment. This is important to avoid caching a module's contents and serving the wrong content when a different module is requested.

If the `USE_I18N` setting is set to `True`, the per-site middleware cache will respect the active language. If you use the `{% cache %}` template tag you have to use one of the translation-specific variables available in templates to achieve the same result, such as `{% cache 600 name request.LANGUAGE_CODE %}`.

Caching views

You can cache the output of individual views using the `cache_page` decorator located at `django.views.decorators.cache`. The decorator requires a `timeout` argument (in seconds).

Let's use it in our views. Edit the `urls.py` file of the `students` application and add the following import:

```
from django.views.decorators.cache import cache_page
```

Then, apply the `cache_page` decorator to the `student_course_detail` and `student_course_detail_module` URL patterns, as follows:

```
path('course/<pk>/',
     cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
     name='student_course_detail'),

path('course/<pk>/<module_id>/',
     cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
     name='student_course_detail_module'),
```

Now, the result for the `StudentCourseDetailView` is cached for 15 minutes.

The per-view cache uses the URL to build the cache key. Multiple URLs pointing to the same view will be cached separately.

Using the per-site cache

This is the highest-level cache. It allows you to cache your entire site.

To allow the per-site cache, edit the `settings.py` file of your project and add the `UpdateCacheMiddleware` and `FetchFromCacheMiddleware` classes to the `MIDDLEWARE` setting, as follows:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    # ...
]
```

Remember that middlewares are executed in the given order during the request phase, and in reverse order during the response phase. `UpdateCacheMiddleware` is placed before `CommonMiddleware` because it runs during response time, when middlewares are executed in reverse order. `FetchFromCacheMiddleware` is placed after `CommonMiddleware` intentionally because it needs to access request data set by the latter.

Then, add the following settings to the `settings.py` file:

```
CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 60 * 15  # 15 minutes
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

In these settings, we use the default cache for our cache middleware and we set the global cache timeout to `15` minutes. We also specify a prefix for all cache keys to avoid collisions in case we use the same Memcached backend for multiple projects. Our site will now cache and return cached content for all GET requests.

We have done this to test the per-site cache functionality. However, the per-site cache is not suitable for us, since the course management views need to show updated data to instantly reflect any changes. The best approach to follow in our project is to cache the templates or views that are used to display course contents to students.

We have seen an overview of the methods provided by Django to cache data. You should define your cache strategy wisely and prioritize the most expensive QuerySets or calculations.

Summary

In this chapter, we created public views for the courses and you have built a system for students to register and enroll in courses. We installed Memcached and implemented different cache levels.

In the next chapter, we will build a RESTful API for your project.

Building an API

In the previous chapter, you built a system of student registration and enrollment in courses. You created views to display course contents and learned how to use Django's cache framework. In this chapter, you will learn how to do the following:

- Build a RESTful API
- Handle authentication and permissions for API views
- Create API view sets and routers

Building a RESTful API

You might want to create an interface for other services to interact with your web application. By building an API, you can allow third parties to consume information and operate with your application programmatically.

There are several ways you can structure your API but following REST principles is encouraged. The **REST** architecture comes from **Representational State Transfer**. RESTful APIs are resource-based. Your models represent resources and HTTP methods such as `GET`, `POST`, `PUT`, or `DELETE` are used to retrieve, create, update, or delete objects. HTTP response codes are also used in this context. Different HTTP response codes are returned to indicate the result of the HTTP request, for example, 2XX response codes for success, 4XX for errors, and so on.

The most common formats to exchange data in RESTful APIs are JSON and XML. We will build a REST API with JSON serialization for our project. Our API will provide the following functionality:

- Retrieve subjects
- Retrieve available courses
- Retrieve course contents
- Enroll in a course

We can build an API from scratch with Django by creating custom views. However, there are several third-party modules that simplify creating an API for your project, the most popular among them being Django REST framework.

Installing Django REST framework

Django REST framework allows you to easily build REST APIs for your project. You can find all information about REST framework at https://www.django-rest-framework.org/.

Open the shell and install the framework with the following command:

```
pip install djangorestframework==3.8.2
```

Edit the settings.py file of the educa project and add rest_framework to the INSTALLED_APPS setting to activate the application, as follows:

```
INSTALLED_APPS = [
    # ...
    'rest_framework',
]
```

Then, add the following code to the settings.py file:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES':
'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

You can provide a specific configuration for your API using the REST_FRAMEWORK setting. REST framework offers a wide range of settings to configure default behaviors. The DEFAULT_PERMISSION_CLASSES setting specifies the default permissions to read, create, update, or delete objects. We set `DjangoModelPermissionsOrAnonReadOnly` as the only default permission class. This class relies on Django's permissions system to allow users to create, update, or delete objects, while providing read-only access for anonymous users. You will learn more about permissions later in the *Adding permissions to views* section.

For a complete list of available settings for REST framework, you can visit `https://www.django-rest-framework.org/api-guide/settings/`.

Defining serializers

After setting up REST framework, we need to specify how our data will be serialized. Output data has to be serialized in a specific format, and input data will be de-serialized for processing. The framework provides the following classes to build serializers for single objects:

- `Serializer`: Provides serialization for normal Python class instances
- `ModelSerializer`: Provides serialization for model instances
- `HyperlinkedModelSerializer`: The same as `ModelSerializer`, but it represents object relationships with links rather than primary keys

Let's build our first serializer. Create the following file structure inside the `courses` application directory:

```
api/
    __init__.py
    serializers.py
```

We will build all the API functionality inside the `api` directory to keep everything well organized. Edit the `serializers.py` file and add the following code:

```
from rest_framework import serializers
from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Subject
        fields = ['id', 'title', 'slug']
```

This is the serializer for the `Subject` model. Serializers are defined in a similar fashion to Django's `Form` and `ModelForm` classes. The `Meta` class allows you to specify the model to serialize and the fields to be included for serialization. All model fields will be included if you don't set a `fields` attribute.

Let's try our serializer. Open the command line and start the Django shell with the following command:

```
python manage.py shell
```

Run the following code:

```
>>> from courses.models import Subject
>>> from courses.api.serializers import SubjectSerializer
>>> subject = Subject.objects.latest('id')
>>> serializer = SubjectSerializer(subject)
>>> serializer.data
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

In this example, we get a `Subject` object, create an instance of `SubjectSerializer`, and access the serialized data. You can see that the model data is translated into Python native data types.

Understanding parsers and renderers

The serialized data has to be rendered in a specific format before you return it in an HTTP response. Likewise, when you get an HTTP request, you have to parse the incoming data and de-serialize it before you can operate with it. REST framework includes renderers and parsers to handle that.

Let's see how to parse incoming data. Execute the following code in the Python shell:

```
>>> from io import BytesIO
>>> from rest_framework.parsers import JSONParser
>>> data = b'{"id":4,"title":"Programming","slug":"programming"}'
>>> JSONParser().parse(BytesIO(data))
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

Given a JSON string input, you can use the `JSONParser` class provided by REST framework to convert it to a Python object.

REST framework also includes `Renderer` classes that allow you to format API responses. The framework determines which renderer to use through content negotiation. It inspects the request's `Accept` header to determine the expected content type for the response. Optionally, the renderer is determined by the format suffix of the URL. For example, accessing will trigger the `JSONRenderer` in order to return a JSON response.

Go back to the shell and execute the following code to render the `serializer` object from the previous serializer example:

```
>>> from rest_framework.renderers import JSONRenderer
>>> JSONRenderer().render(serializer.data)
```

You will see the following output:

```
b'{"id":4,"title":"Programming","slug":"programming"}'
```

We use the `JSONRenderer` to render the serialized data into JSON. By default, REST framework uses two different renderers: `JSONRenderer` and `BrowsableAPIRenderer`. The latter provides a web interface to easily browse your API. You can change the default renderer classes with the `DEFAULT_RENDERER_CLASSES` option of the `REST_FRAMEWORK` setting.

You can find more information about renderers and parsers at `https://www.django-rest-framework.org/api-guide/renderers/` and `https://www.django-rest-framework.org/api-guide/parsers/`, respectively.

Building list and detail views

REST framework comes with a set of generic views and mixins that you can use to build your API views. These provide functionality to retrieve, create, update, or delete model objects. You can see all generic mixins and views provided by REST framework at `https://www.django-rest-framework.org/api-guide/generic-views/`.

Let's create list and detail views to retrieve `Subject` objects. Create a new file inside the `courses/api/` directory and name it `views.py`. Add the following code to it:

```
from rest_framework import generics
from ..models import Subject
from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer
```

In this code, we are using the generic `ListAPIView` and `RetrieveAPIView` views of REST framework. We include a `pk` URL parameter for the detail view to retrieve the object for the given primary key. Both views have the following attributes:

- `queryset`: The base `QuerySet` to use to retrieve objects
- `serializer_class`: The class to serialize objects

Let's add URL patterns for our views. Create a new file inside the `courses/api/` directory, name it `urls.py`, and make it look as follows:

```
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'),
]
```

Edit the main `urls.py` file of the `educa` project and include the API patterns as follows:

```
urlpatterns = [
    # ...
    path('api/', include('courses.api.urls', namespace='api')),
]
```

We use the `api` namespace for our API URLs. Ensure that your server is running with the command `python manage.py runserver`. Open the shell and retrieve the URL `http://127.0.0.1:8000/api/subjects/` with `curl` as follows:

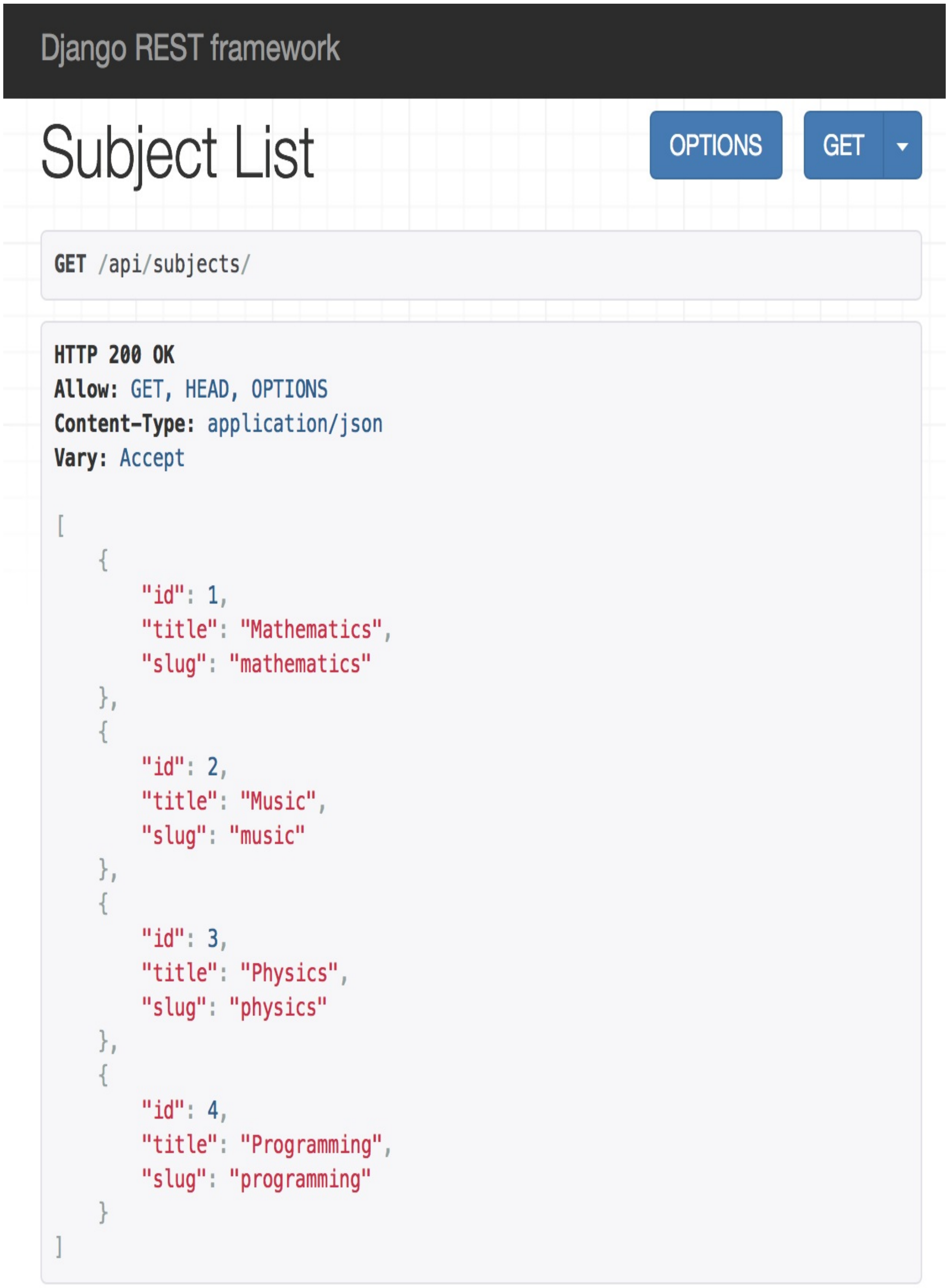
```
curl http://127.0.0.1:8000/api/subjects/
```

You will get a response similar to the following one:

```
[
    {"id":1,"title":"Mathematics","slug":"mathematics"},
    {"id":2,"title":"Music","slug":"music"},
    {"id":3,"title":"Physics","slug":"physics"},
    {"id":4,"title":"Programming","slug":"programming"}
]
```

The HTTP response contains a list of `Subject` objects in JSON format. If your operating system doesn't come with `curl` installed, you can download it from `https://curl.haxx.se/dlwiz/`. Instead of `curl`, you can also use any other tool to send custom HTTP requests, such as a browser extension, such as Postman, which you can get at `https://www.getpostman.com/`.

Open `http://127.0.0.1:8000/api/subjects/` in your browser. You will see REST framework's browsable API as follows:



Subject List

```
GET /api/subjects/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "Mathematics",
        "slug": "mathematics"
    },
    {
        "id": 2,
        "title": "Music",
        "slug": "music"
    },
    {
        "id": 3,
        "title": "Physics",
        "slug": "physics"
    },
    {
        "id": 4,
        "title": "Programming",
        "slug": "programming"
    }
]
```

This HTML interface is provided by the `BrowsableAPIRenderer` renderer. It displays the result headers and content and allows you to perform requests. You can also access the API detail view for a `Subject` object by including its ID in the URL. Open `http://127.0.0.1:8000/api/subjects/1/` in your browser. You will see a single `Subject` object rendered in JSON format.

Creating nested serializers

We are going to create a serializer for the `Course` model. Edit the `api/serializers.py` file of the `courses` application and add the following code to it:

```
from ..models import Course

class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview',
                  'created', 'owner', 'modules']
```

Let's take a look at how a `Course` object is serialized. Open the shell, run `python manage.py shell`, and run the following code:

```
>>> 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)
```

You will get a JSON object with the fields we included in `CourseSerializer`. You can see that the related objects of the `modules` manager are serialized as a list of primary keys, as follows:

```
"modules": [6, 7, 9, 10]
```

We want to include more information about each module, so we need to serialize `Module` objects and nest them. Modify the previous code of the `api/serializers.py` file of the `courses` application to make it look as follows:

```
from rest_framework import serializers
from ..models import 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']
```

We define `ModuleSerializer` to provide serialization for the `Module` model. Then we add a `modules` attribute to `CourseSerializer` to nest the `ModuleSerializer` serializer. We set `many=True` to indicate that we are serializing multiple objects. The `read_only` parameter indicates that this field is read-only and should not be included in any input to create or update objects.

Open the shell and create an instance of `CourseSerializer` again. Render the serializer's `data` attribute with `JSONRenderer`. This time, the listed modules are being serialized with the nested `ModuleSerializer` serializer, as follows:

```
"modules": [
    {
        "order": 0,
        "title": "Introduction to overview",
        "description": "A brief overview about the Web Framework."
    },
    {
        "order": 1,
        "title": "Configuring Django",
        "description": "How to install Django."
    },
    ...
]
```

You can read more about serializers at `https://www.django-rest-framework.o` `rg/api-guide/serializers/.`

Building custom views

REST framework provides an `APIView` class, which builds API functionality on top of Django's `View` class. The `APIView` class differs from `View` in using REST framework's custom `Request` and `Response` objects and handling `APIException` exceptions to return the appropriate HTTP responses. It also has a built-in authentication and authorization system to manage access to views.

We are going to create a view for users to enroll in courses. Edit the `api/views.py` file of the `courses` application and add the following code to it:

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from ..models import Course

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})
```

The `CourseEnrollView` view handles user enrollment in courses. The preceding code is as follows:

1. We create a custom view that subclasses `APIView`.
2. We define a `post()` method for POST actions. No other HTTP method will be allowed for this view.
3. We expect a `pk` URL parameter containing the ID of a course. We retrieve the course by the given `pk` parameter and raise a `404` exception if it's not found.

4. We add the current user to the `students` many-to-many relationship of the `Course` object and return a successful response.

Edit the `api/urls.py` file and add the following URL pattern for the `CourseEnrollView` view:

```
path('courses/<pk>/enroll/',
     views.CourseEnrollView.as_view(),
     name='course_enroll'),
```

Theoretically, we could now perform a POST request to enroll the current user in a course. However, we need to be able to identify the user and prevent unauthenticated users from accessing this view. Let's see how API authentication and permissions work.

Handling authentication

REST framework provides authentication classes to identify the user performing the request. If authentication is successful, the framework sets the authenticated `User` object in `request.user`. If no user is authenticated, an instance of Django's `AnonymousUser` is set instead.

REST framework provides the following authentication backends:

- `BasicAuthentication`: This is HTTP basic authentication. The user and password are sent by the client in the `Authorization` HTTP header encoded with Base64. You can learn more about it at `https://en.wikipedia.org/wiki/Basic_access_authentication`.
- `TokenAuthentication`: This is token-based authentication. A `Token` model is used to store user tokens. Users include the token in the `Authorization` HTTP header for authentication.
- `SessionAuthentication`: This one uses Django's session backend for authentication. This backend is useful to perform authenticated AJAX requests to the API from your website's frontend.
- `RemoteUserAuthentication`: This allows you to delegate authentication to your web server, which sets a `REMOTE_USER` environment variable.

You can build a custom authentication backend by subclassing the `BaseAuthentication` class provided by REST framework and overriding the `authenticate()` method.

You can set authentication on a per-view basis, or set it globally with the `DEFAULT_AUTHENTICATION_CLASSES` setting.

> *Authentication only identifies the user performing the request. It won't allow or deny access to views. You have to use permissions to restrict access to views.*

You can find all the information about authentication at `https://www.django-rest-framework.org/api-guide/authentication/`.

Let's add `BasicAuthentication` to our view. Edit the `api/views.py` file of the `courses` application and add an `authentication_classes` attribute to `CourseEnrollView` as follows:

```
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    # ...
```

Users will be identified by the credentials set in the `Authorization` header of the HTTP request.

Adding permissions to views

REST framework includes a permission system to restrict access to views. Some of the built-in permissions of REST framework are:

- `AllowAny`: Unrestricted access, regardless of if a user is authenticated or not.
- `IsAuthenticated`: Allows access to authenticated users only.
- `IsAuthenticatedOrReadOnly`: Complete access to authenticated users. Anonymous users are only allowed to execute read methods such as `GET`, `HEAD`, or `OPTIONS`.
- `DjangoModelPermissions`: Permissions tied to `django.contrib.auth`. The view requires a `queryset` attribute. Only authenticated users with model permissions assigned are granted permission.
- `DjangoObjectPermissions`: Django permissions on a per-object basis.

If users are denied permission, they will usually get one of the following HTTP error codes:

- `HTTP 401`: Unauthorized
- `HTTP 403`: Permission denied

You can read more information about permissions at `https://www.django-rest-framework.org/api-guide/permissions/`.

Edit the `api/views.py` file of the `courses` application and add a `permission_classes` attribute to `CourseEnrollView` as follows:

```
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    permission_classes = (IsAuthenticated,)
    # ...
```

We include the `IsAuthenticated` permission. This will prevent anonymous users from accessing the view. Now we can perform a POST request to our new API method.

Make sure the development server is running. Open the shell and run the following command:

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

You will get the following response:

```
HTTP/1.1 401 Unauthorized
...
{"detail": "Authentication credentials were not provided."}
```

We get a `401` HTTP code as expected, since we are not authenticated. Let's use basic authentication with one of our users. Run the following command, replacing `student:password` with the credentials of an existing user:

```
curl -i -X POST -u student:password
http://127.0.0.1:8000/api/courses/1/enroll/
```

You will get the following response:

```
HTTP/1.1 200 OK
```

```
...
{"enrolled": true}
```

You can access the administration site and check that the user is now enrolled in the course.

Creating view sets and routers

`ViewSets` allow you to define the interactions of your API and let REST framework build the URLs dynamically with a `Router` object. By using view sets, you can avoid repeating logic for multiple views. View sets include actions for the typical create, retrieve, update, delete operations, which are `list()`, `create()`, `retrieve()`, `update()`, `partial_update()`, and `destroy()`.

Let's create a view set for the `Course` model. Edit the `api/views.py` file and add the following code to it:

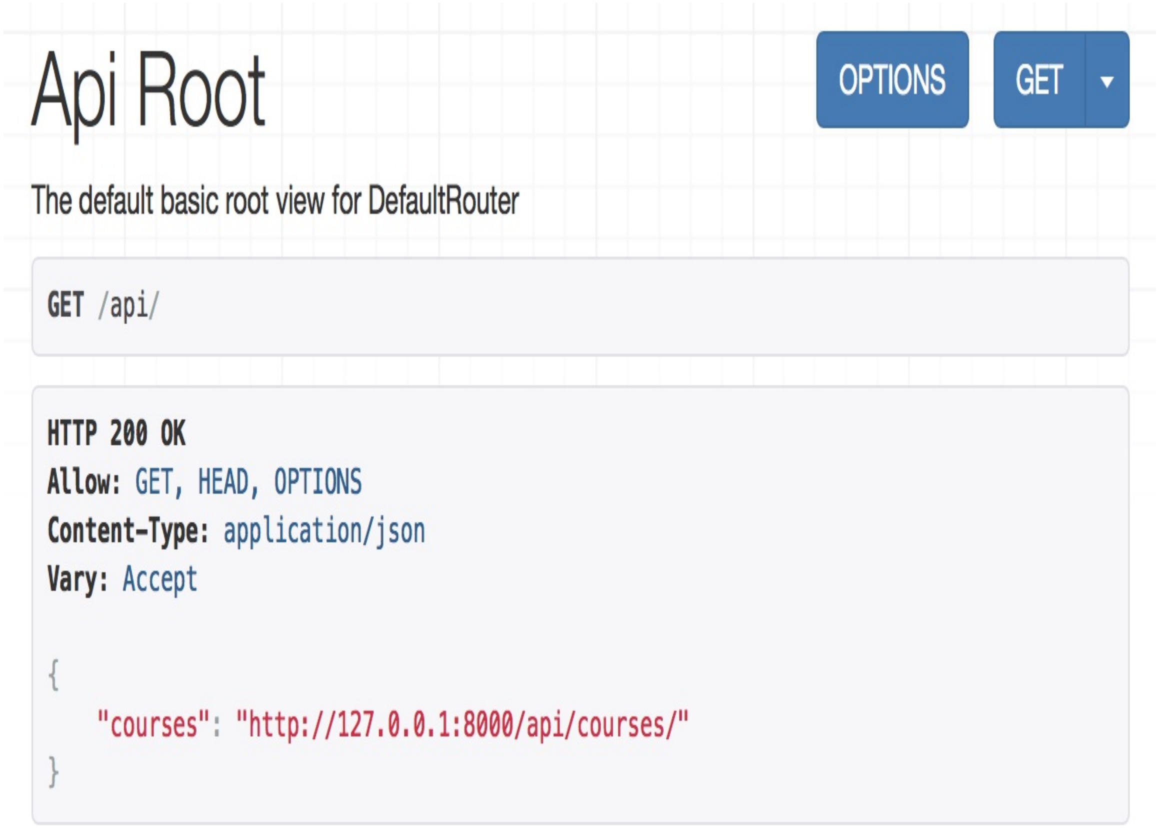
```
from rest_framework import viewsets
from .serializers import CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
```

We subclass `ReadOnlyModelViewSet`, which provides the read-only actions `list()` and `retrieve()` to both list objects or retrieve a single object. Edit the `api/urls.py` file and create a router for our view set as follows:

```
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)),
]
```

We create a `DefaultRouter` object and register our view set with the `courses` prefix. The router takes charge of generating URLs automatically for our view set.

Open `http://127.0.0.1:8000/api/` in your browser. You will see that the router lists all view sets in its base URL, as shown in the following screenshot:



```
GET /api/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "courses": "http://127.0.0.1:8000/api/courses/"
}
```

You can access `http://127.0.0.1:8000/api/courses/` to retrieve the list of courses.

You can learn more about view sets at `https://www.django-rest-framework.org/api-guide/viewsets/`. You can also find more information about routers at `https://www.django-rest-framework.org/api-guide/routers/`.

Adding additional actions to view sets

You can add extra actions to view sets. Let's change our previous `CourseEnrollView` view into a custom view set action. Edit the `api/views.py` file and modify the `CourseViewSet` class to look as follows:

```
from rest_framework.decorators import detail_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

    @detail_route(methods=['post'],
                  authentication_classes=[BasicAuthentication],
                  permission_classes=[IsAuthenticated])
    def enroll(self, request, *args, **kwargs):
        course = self.get_object()
        course.students.add(request.user)
        return Response({'enrolled': True})
```

We add a custom `enroll()` method that represents an additional action for this view set. The preceding code is as follows:

1. We use the `detail_route` decorator of the framework to specify that this is an action to be performed on a single object.
2. The decorator allows us to add custom attributes for the action. We specify that only the `post` method is allowed for this view and set the authentication and permission classes.
3. We use `self.get_object()` to retrieve the `Course` object.
4. We add the current user to the `students` many-to-many relationship and return a custom success response.

Edit the `api/urls.py` file and remove the following URL, since we don't need it anymore:

```
path('courses/<pk>/enroll/',
     views.CourseEnrollView.as_view(),
     name='course_enroll'),
```

Then edit the `api/views.py` file and remove the `CourseEnrollView` class.

The URL to enroll in courses is now automatically generated by the router. The URL remains the same, since it's built dynamically using our action name `enroll`.

Creating custom permissions

We want students to be able to access the contents of the courses they are enrolled in. Only students enrolled in a course should be able to access its contents. The best way to do this is with a custom permission class. Django provides a `BasePermission` class that allows you to define the following methods:

- `has_permission()`: View-level permission check
- `has_object_permission()`: Instance-level permission check

These methods should return `True` to grant access or `False` otherwise. Create a new file inside the `courses/api/` directory and name it `permissions.py`. Add the following code to it:

```
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()
```

We subclass the `BasePermission` class and override the `has_object_permission()`. We check that the user performing the request is present in the `students` relationship of the `Course` object. We are going to use the `IsEnrolled` permission next.

Serializing course contents

We need to serialize course contents. The `Content` model includes a generic foreign key that allows us to associate objects of different content models. Yet, we have added a common `render()` method for all content models in the previous chapter. We can use this method to provide rendered contents to our API.

Edit the `api/serializers.py` file of the `courses` application and add the following code to it:

```
from ..models import Content

class ItemRelatedField(serializers.RelatedField):
    def to_representation(self, value):
        return value.render()

class ContentSerializer(serializers.ModelSerializer):
    item = ItemRelatedField(read_only=True)

    class Meta:
        model = Content
        fields = ['order', 'item']
```

In this code, we define a custom field by subclassing the `RelatedField` serializer field provided by REST framework and overriding the `to_representation()` method. We define the `ContentSerializer` serializer for the `Content` model and use the custom field for the `item` generic foreign key.

We need an alternate serializer for the `Module` model that includes its contents, and an extended `Course` serializer as well. Edit the `api/serializers.py` file and add the following code to it:

```
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']
```

Let's create a view that mimics the behavior of the `retrieve()` action, but it includes the course contents. Edit the `api/views.py` file and add the following method to the `CourseViewSet` class:

```
from .permissions import IsEnrolled
from .serializers import CourseWithContentsSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    # ...
    @detail_route(methods=['get'],
                  serializer_class=CourseWithContentsSerializer,
                  authentication_classes=[BasicAuthentication],
                  permission_classes=[IsAuthenticated,
                                      IsEnrolled])
    def contents(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

The description of this method is as follows:

- We use the `detail_route` decorator to specify that this action is performed on a single object.
- We specify that only the `GET` method is allowed for this action.
- We use the new `CourseWithContentsSerializer` serializer class that includes rendered course contents.

- We use both the `IsAuthenticated` and our custom `IsEnrolled` permissions. By doing so, we make sure that only users enrolled in the course are able to access its contents.
- We use the existing `retrieve()` action to return the `Course` object.

Open `http://127.0.0.1:8000/api/courses/1/contents/` in your browser. If you access the view with the right credentials, you will see that each module of the course includes the rendered HTML for course contents, as follows:

```
{
    "order": 0,
    "title": "Introduction to Django",
    "description": "Brief introduction to the Django Web Framework.",
    "contents": [
        {
            "order": 0,
            "item": "<p>Meet Django. Django is a high-level
            Python Web framework
            ...</p>"
        },
        {
            "order": 1,
            "item": "\n<iframe width=\"480\" height=\"360\"
            src=\"http://www.youtube.com/embed/bgV39DlmZ2U?
            wmode=opaque\"
            frameborder=\"0\" allowfullscreen></iframe>\n"
        }
    ]
}
```

You have built a simple API that allows other services to access the course application programmatically. REST framework also allows you to handle creating and editing objects with the `ModelViewSet` view set. We have covered the main aspects of Django REST framework, but you will find further information about its features in its extensive documentation at `https://www.django-rest-framework.org/`.

Summary

In this chapter, you created a RESTful API for other services to interact with your web application.