New code is highlighted in bold:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.auth import views as auth_views

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

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

You 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, you list the courses created by the current user. You 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 belonging 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:

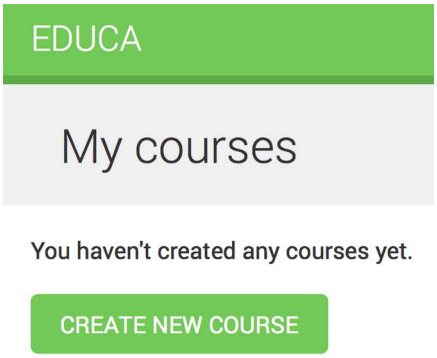


Figure 13.3: The instructor courses page with no courses

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 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, you check whether an `object` variable is in the context. If `object` exists in the context, you know that you are updating an existing course, and you use it in the page title. Otherwise, you 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:

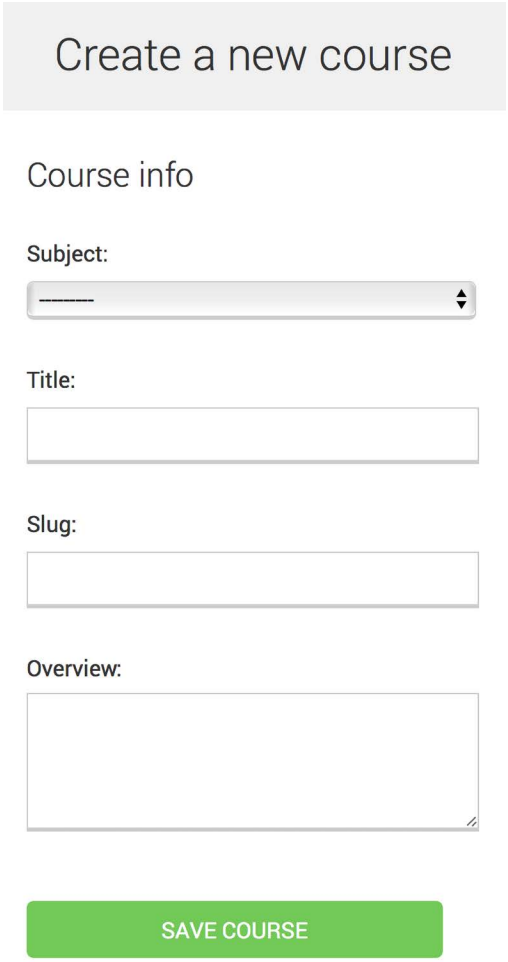


Figure 13.4: The form to create a new course

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:

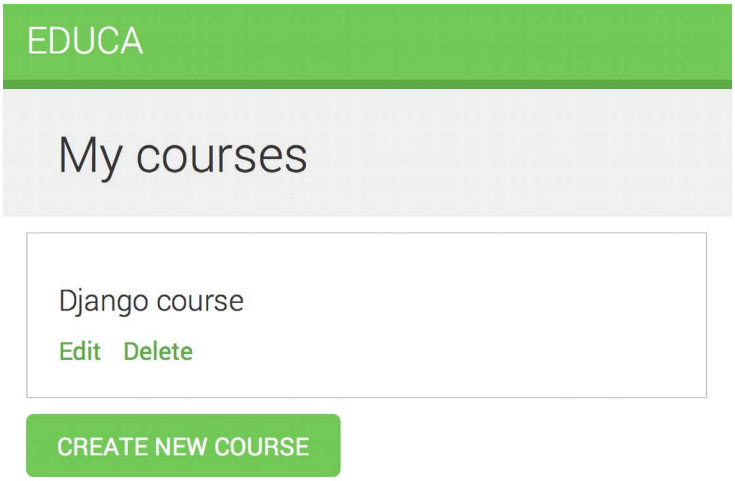


Figure 13.5: The instructor courses page with one course

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" 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 the course list in the browser and click the **Delete** link of your course. You should see the following confirmation page:

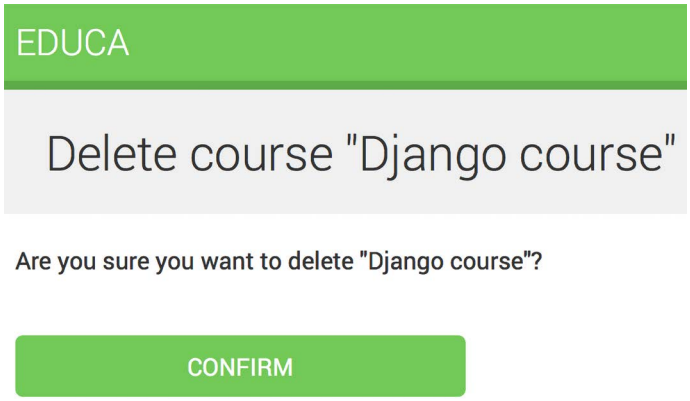


Figure 13.6: The delete course confirmation page

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, you need to provide them with a CMS to add course modules and their contents. You will start by managing course modules.

Managing course modules and their contents

You are going to build a system to manage course modules and their contents. You 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 their contents will have to follow a specific order and you 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/4.1/topics/forms/formsets/` and about model formsets at `https://docs.djangoproject.com/en/4.1/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. You 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 you to build a model formset dynamically for the `Module` objects related to a `Course` object.

You use the following parameters to build the formset:

- `fields`: The fields that will be included in each form of the formset.
- `extra`: Allows you 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 that 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().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, you implement the following methods:

- `get_formset()`: You define this method to avoid repeating the code to build the formset. You 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, you use the `get_object_or_404()` shortcut function to get the `Course` object for the given `id` parameter that belongs to the current user. You include this code in the `dispatch()` method because you need to retrieve the course for both `GET` and `POST` requests. You save it into the `course` attribute of the view to make it accessible to other methods.
- `get()`: Executed for `GET` requests. You 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, you perform the following actions:

    1. You build a `ModuleFormSet` instance using the submitted data.
    2. You execute the `is_valid()` method of the formset to validate all of its forms.
    3. If the formset is valid, you 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, you redirect users to the `manage_course_list` URL. If the formset is not valid, you 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 method="post">
      {{ formset }}
      {{ formset.management_form }}
      {% csrf_token %}
      <input type="submit" value="Save modules">
    </form>
  </div>
{% endblock %}
```

In this template, you create a `<form>` HTML element in which you include `formset`. You 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 that 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>
```

You 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:

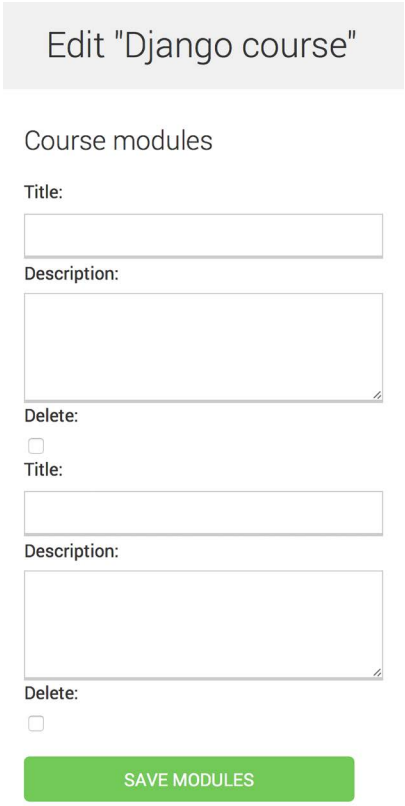


Figure 13.7: The course edit page, including the formset for course modules

The formset includes a form for each `Module` object contained in the course. After these, two empty extra forms are displayed because you 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, you need a way to add content to course modules. You have four different types of content: text, video, image, and file. You could consider creating four different views to create content, with one for each model. However, you are going to take a more generic approach and create a view that handles creating or updating the 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().dispatch(request, module_id, model_name, id)
```

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

- `get_model()`: Here, you check that the given model name is one of the four content models: `Text`, `Video`, `Image`, or `File`. Then, you 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, you return `None`.
- `get_form()`: You build a dynamic form using the `modelform_factory()` function of the form's framework. Since you are going to build a form for the `Text`, `Video`, `Image`, and `File` models, you 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, you 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. You build the model form for the `Text`, `Video`, `Image`, or `File` instance that is being updated. Otherwise, you 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. You build the model form, passing any submitted data and files to it. Then, you validate it. If the form is valid, you create a new object and assign `request.user` as its owner before saving it to the database. You check for the `id` parameter. If no ID is provided, you know the user is creating a new object instead of updating an existing one. If this is a new object, you create a `Content` object for the given module and associate the new content with 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 new content
  {% endif %}
{% endblock %}

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

```
    </form>
  </div>
{% endblock %}
```

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

You 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.

Then open the Python shell with the following command:

```
python manage.py shell
```

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 with the one you obtained before. You will see the form to create an `Image` object, as follows:

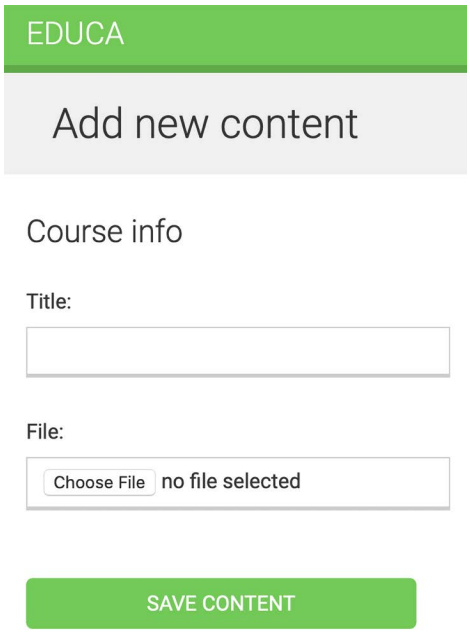


Figure 13.8: The course add new image content form

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

You also need a view for deleting content. 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. 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 content easily.

Managing modules and their contents

You have built views to create, edit, and delete course modules and their contents. Next, you need a view to display all modules for a course and list the contents of 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 the `ModuleContentListView` view. 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 %}
```

Make sure that no template tag is split into multiple lines.

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

You want to know which type of object each of the `item` objects is: `Text`, `Video`, `Image`, or `File`. You need the model name to build the URL to edit the object. Besides this, you could display each item in the template differently based on the type of content it is. You can get the model name 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. You 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. You 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, find 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>
```

In the preceding code, you display the item model name in the template and also 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 there are any.

Stop the development server and run it again using the command:

```
python manage.py runserver
```

By stopping and running the development server, you make sure that the `course` template tags file gets loaded.

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:

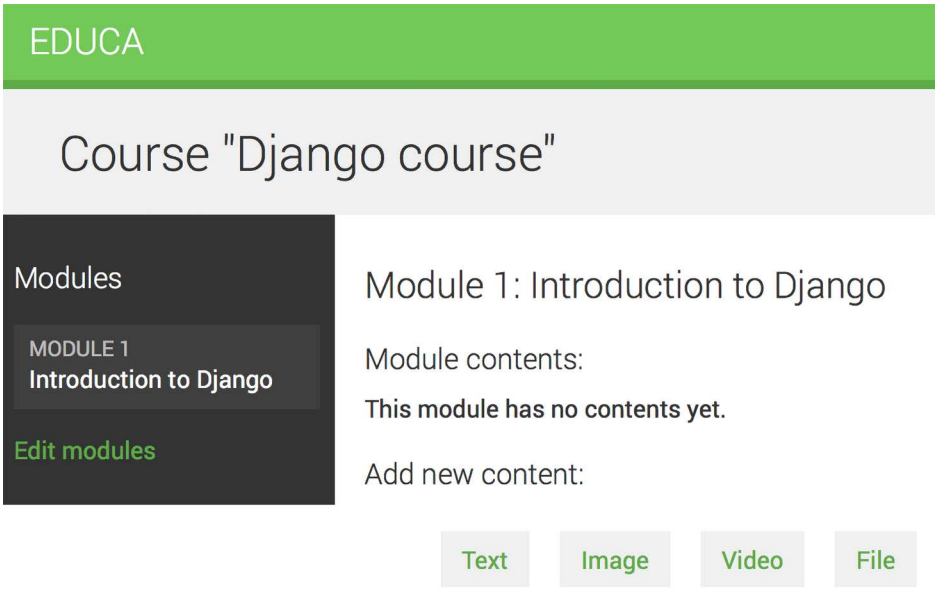


Figure 13.9: The page to manage course module contents

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 new text, video, image, or file content for the module being displayed.

Add a couple of different types of content to the module and look at the result. Module contents will appear below **Module contents:**

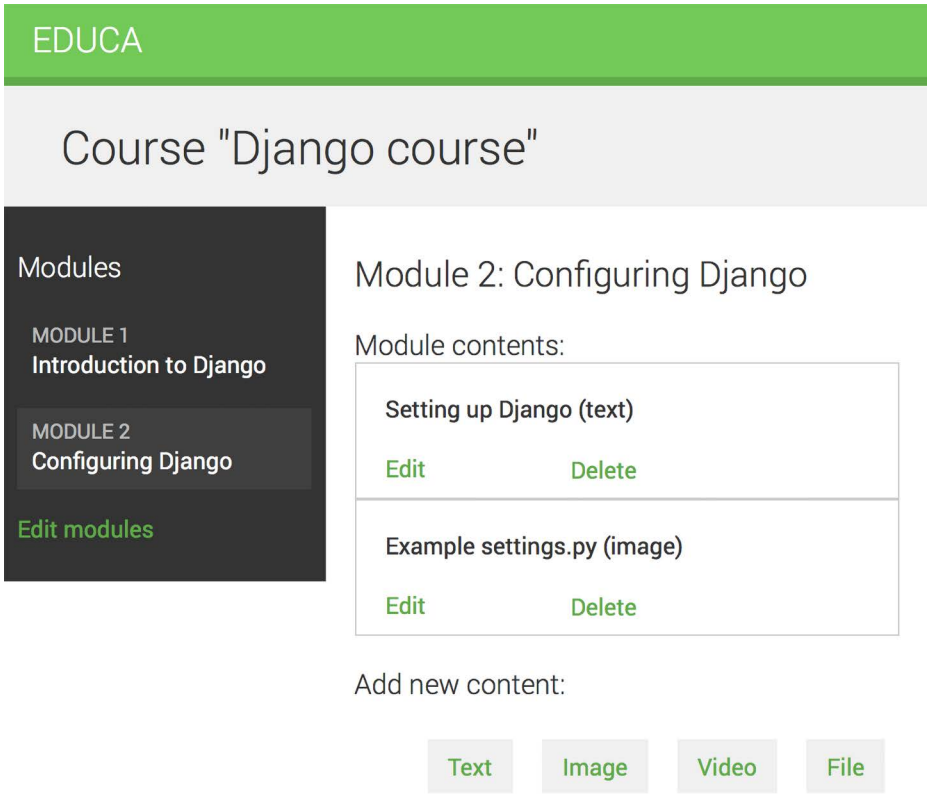


Figure 13.10: Managing different module contents

Next, we will allow course instructors to reorder modules and module contents with a simple drag-and-drop functionality.

Reordering modules and their contents

We will implement a JavaScript drag-and-drop functionality to let course instructors reorder the modules of a course by dragging them.

To implement this feature, we will use the HTML5 Sortable library, which simplifies the process of creating sortable lists using the native HTML5 Drag and Drop API.

When users finish dragging a module, you will use the JavaScript Fetch API to send an asynchronous HTTP request to the server that stores the new module order.

You can read more information about the HTML5 Drag and Drop API at `https://www.w3schools.com/html/html5_draganddrop.asp`. You can find examples built with the HTML5 Sortable library at `https://lukasoppermann.github.io/html5sortable/`. Documentation for the HTML5 Sortable library is available at `https://github.com/lukasoppermann/html5sortable`.

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/`.

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

- `CsrfExemptMixin`: Used to avoid checking the **cross-site request forgery** (**CSRF**) token in the `POST` requests. You need this to perform AJAX `POST` requests without the need to pass 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==1.15.0
```

You need a view that receives the new order of module IDs encoded in JSON and updates the order accordingly. 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, which allows you to update the order of course modules.

You 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, you need to implement the drag-and-drop functionality in the template. We will use the HTML5 Sortable library, which simplifies the creation of sortable elements using the standard HTML Drag and Drop API.

Edit the `base.html` template located in the `templates/` directory of the `courses` application and add the following block highlighted in bold:

```
{% load static %}
<!DOCTYPE html>
<html>
  <head>
    # ...
  </head>
  <body>
    <div id="header">
      # ...
    </div>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
    {% block include_js %}
    {% endblock %}
    <script>
      document.addEventListener('DOMContentLoaded', (event) => {
        // DOM loaded
        {% block domready %}
        {% endblock %}
      })
    </script>
  </body>
</html>
```

This new block named `include_js` will allow you to insert JavaScript files in any template that extends the `base.html` template.

Next, edit the `courses/manage/module/content_list.html` template and add the following code highlighted in bold to the bottom of the template:

```
# ...
{% block content %}
  # ...
{% endblock %}

{% block include_js %}
  <script src="https://cdnjs.cloudflare.com/ajax/libs/html5sortable/0.13.3/
html5sortable.min.js"></script>
{% endblock %}
```

In this code, you load the HTML5 Sortable library from a public CDN. Remember you loaded a JavaScript library from a content delivery network before in *Chapter 6, Sharing Content on Your Website*.

Now add the following `domready` block highlighted in bold to the `courses/manage/module/content_list.html` template:

```
# ...
{% block content %}
  # ...
{% endblock %}

{% block include_js %}
  <script src="https://cdnjs.cloudflare.com/ajax/libs/html5sortable/0.13.3/
html5sortable.min.js"></script>
{% endblock %}

{% block domready %}
  var options = {
      method: 'POST',
      mode: 'same-origin'
  }
  const moduleOrderUrl = '{% url "module_order" %}';
{% endblock %}
```

In these new lines, you add JavaScript code to the `{% block domready %}` block that was defined in the event listener for the `DOMContentLoaded` event in the `base.html` template. This guarantees that your JavaScript code will be executed once the page has been loaded. With this code, you define the options for the HTTP request to reorder modules that you will implement next. You will send a `POST` request using the Fetch API to update the module order. The `module_order` URL path is built and stored in the JavaScript constant `moduleOrderUrl`.

Add the following code highlighted in bold to the `domready` block:

```
{% block domready %}
  var options = {
      method: 'POST',
      mode: 'same-origin'
  }
  const moduleOrderUrl = '{% url "module_order" %}';

  sortable('#modules', {
    forcePlaceholderSize: true,
    placeholderClass: 'placeholder'
  });
{% endblock %}
```

In the new code, you define a `sortable` element for the HTML element with `id="modules"`, which is the module list in the sidebar. Remember that you use a CSS selector `#` to select the element with the given `id`. When you start dragging an item, the HTML5 Sortable library creates a placeholder item so that you can easily see where the element will be placed.

You set the `forcePlacehoderSize` option to `true`, to force the placeholder element to have a height, and you use the `placeholderClass` to define the CSS class for the placeholder element. You use the class named `placeholder` that is defined in the `css/base.css` static file loaded in the `base.html` template.

Open `http://127.0.0.1:8000/course/mine/` in your browser and click on **Manage contents** for any course. Now you can drag and drop the course modules in the left sidebar, as in *Figure 13.11*:

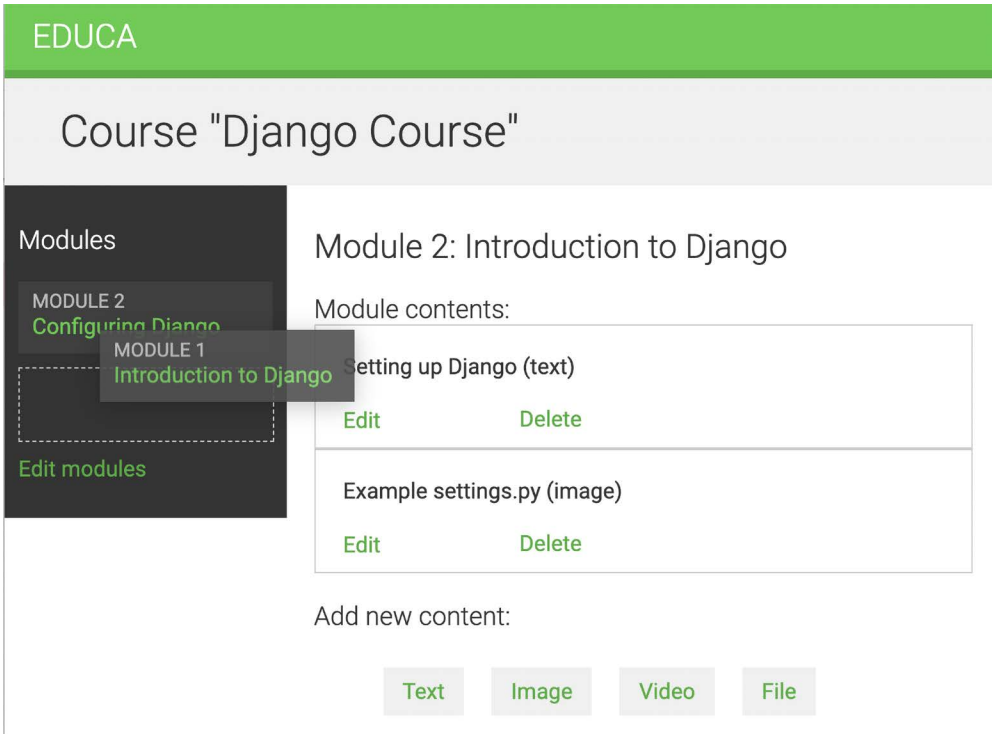


Figure 13.11: Reordering modules with the drag-and-drop functionality

While you drag the element, you will see the placeholder item created by the Sortable library, which has a dashed-line border. The placeholder element allows you to identify the position where the dragged element will be dropped.

When you drag a module to a different position, you need to send an HTTP request to the server to store the new order. This can be done by attaching an event handler to the sortable element and sending a request to the server using the JavaScript Fetch API.

Edit the `domready` block of the `courses/manage/module/content_list.html` template and add the following code highlighted in bold:

```
{% block domready %}
  var options = {
      method: 'POST',
      mode: 'same-origin'
  }
  const moduleOrderUrl = '{% url "module_order" %}';

  sortable('#modules', {
    forcePlaceholderSize: true,
    placeholderClass: 'placeholder'
  })[0].addEventListener('sortupdate', function(e) {

    modulesOrder = {};
    var modules = document.querySelectorAll('#modules li');
    modules.forEach(function (module, index) {
      // update module index
      modulesOrder[module.dataset.id] = index;
      // update index in HTML element
      module.querySelector('.order').innerHTML = index + 1;
      // add new order to the HTTP request options
      options['body'] = JSON.stringify(modulesOrder);

      // send HTTP request
      fetch(moduleOrderUrl, options)
    });
  });

{% endblock %}
```

In the new code, an event listener is created for the `sortupdate` event of the sortable element. The `sortupdate` event is triggered when an element is dropped in a different position. The following tasks are performed in the event function:

1. An empty `modulesOrder` dictionary is created. The keys for this dictionary will be the module IDs, and the values will contain the index of each module.
2. The list elements of the `#modules` HTML element are selected with `document.querySelectorAll()`, using the `#modules li` CSS selector.
3. `forEach()` is used to iterate over each list element.

4. The new index for each module is stored in the `modulesOrder` dictionary. The ID of each module is retrieved from the HTML `data-id` attribute by accessing `module.dataset.id`. You use the ID as the key of the `modulesOrder` dictionary and the new index of the module as the value.
5. The order displayed for each module is updated by selecting the element with the `order` CSS class. Since the index is zero-based and we want to display a one-based index, we add `1` to `index`.
6. A key named `body` is added to the `options` dictionary with the new order contained in `modulesOrder`. The `JSON.stringify()` method converts the JavaScript object into a JSON string. This is the body for the HTTP request to update the module order.
7. The Fetch API is used by creating a `fetch()` HTTP request to update the module order. The view `ModuleOrderView` that corresponds to the `module_order` URL takes care of updating the order of the modules.

You can now drag and drop modules. When you finish dragging a module, an HTTP request is sent to the `module_order` URL to update the order of the modules. If you refresh the page, the latest module order will be kept because it was updated in the database. *Figure 13.12* shows a different order for the modules in the sidebar after sorting them using drag and drop:

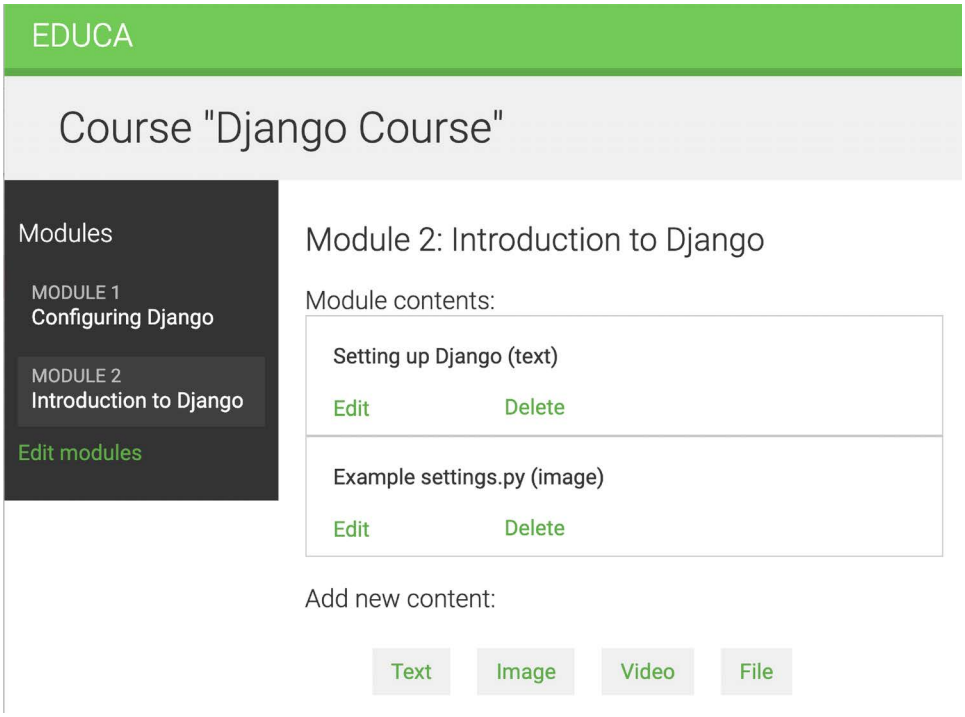


Figure 13.12: New order for modules after reordering them with drag and drop

If you run into any issues, remember to use your browser's developer tools to debug JavaScript and HTTP requests. Usually, you can right-click anywhere on the website to open the contextual menu and click on **Inspect** or **Inspect Element** to access the web developer tools of your browser.

Let's add the same drag-and-drop functionality to allow course instructors to sort module contents as well.

Edit the `domready` block of the `courses/manage/module/content_list.html` template and add the following code highlighted in bold:

```
{% block domready %}

  // ...

  const contentOrderUrl = '{% url "content_order" %}';

  sortable('#module-contents', {
    forcePlaceholderSize: true,
    placeholderClass: 'placeholder'
  })[0].addEventListener('sortupdate', function(e) {

    contentOrder = {};
    var contents = document.querySelectorAll('#module-contents div');
    contents.forEach(function (content, index) {
      // update content index
      contentOrder[content.dataset.id] = index;
      // add new order to the HTTP request options
      options['body'] = JSON.stringify(contentOrder);

      // send HTTP request
      fetch(contentOrderUrl, options)
    });
  });

{% endblock %}
```

In this case, you use the `content_order` URL instead of `module_order` and build the `sortable` functionality on the HTML element with the ID `module-contents`. The functionality is mainly the same as for ordering course modules. In this case, you don't need to update the numbering of the contents because they don't include any visible index.

Now you can drag and drop both modules and module contents, as in *Figure 13.13*:

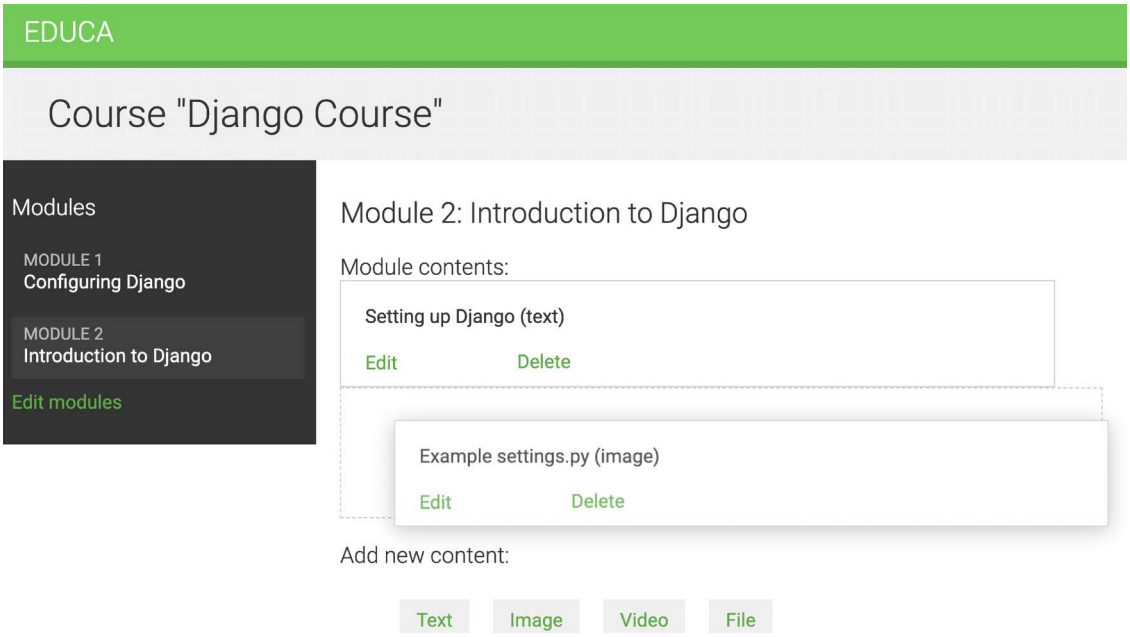


Figure 13.13: Reordering module contents with the drag-and-drop functionality

Great! You built a very versatile content management system for the course instructors.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter13`
- Django mixins documentation – `https://docs.djangoproject.com/en/4.1/topics/class-based-views/mixins/`
- Creating custom permissions – `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#custom-permissions`
- Django formsets – `https://docs.djangoproject.com/en/4.1/topics/forms/formsets/`
- Django model formsets –`https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/#model-formsets`
- HTML5 drag-and-drop API – `https://www.w3schools.com/html/html5_draganddrop.asp`
- HTML5 Sortable library documentation – `https://github.com/lukasoppermann/html5sortable`
- HTML5 Sortable library examples – `https://lukasoppermann.github.io/html5sortable/`
- `django-braces` documentation – `https://django-braces.readthedocs.io/`

Summary

In this chapter, you learned how to use class-based views and mixins to create a content management system. You also worked with groups and permissions to restrict access to your views. You learned how to use formsets and model formsets to manage course modules and their content. You also built a drag-and-drop functionality with JavaScript to reorder course modules and their contents.

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

14

Rendering and Caching Content

In the previous chapter, you used model inheritance and generic relations to create flexible course content models. You implemented a custom model field, and you built a course management system using class-based views. Finally, you created a JavaScript drag-and-drop functionality using asynchronous HTTP requests to order course modules and their contents.

In this chapter, you will build the functionality to access course contents, create a student registration system, and manage student enrollment onto courses. You will also learn how to cache data using the Django cache framework.

In this chapter, you will:

- Create public views for displaying course information
- Build a student registration system
- Manage student enrollment onto courses
- Render diverse content for course modules
- Install and configure Memcached
- Cache content using the Django cache framework
- Use the Memcached and Redis cache backends
- Monitor your Redis server in the Django administration site

Let's start by creating a course catalog for students to browse existing courses and enroll on them.

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter14`.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `pip install -r requirements.txt`.

Displaying courses

For your course catalog, you have to build the following functionalities:

- 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, you perform the following tasks:

1. You retrieve all subjects, using the ORM's `annotate()` method with the `Count()` aggregation function to include the total number of courses for each subject.
2. You retrieve all available courses, including the total number of modules contained in each course.
3. If a subject slug URL parameter is given, you retrieve the corresponding `subject` object and limit the query to the courses that belong to the given subject.
4. You 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. You 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. The view renders the template specified in `template_name`, including the `Course` object in the template context variable `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'),
]
```

You add the `course_list` URL pattern to the main `urls.py` file of the project because you 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'),
```

You 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 of the `courses` application 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 }} course{{ s.total_courses|pluralize }}
            </span>
          </a>
        </li>
      {% endfor %}
    </ul>
  </div>
  <div class="module">
    {% for course in courses %}
      {% with subject=course.subject %}
        <h3>
          <a href="{% url "course_detail" course.slug %}">
            {{ course.title }}
          </a>
        </h3>
```

```
        <p>
          <a href="{% url "course_list_subject" subject.slug %}">{{ subject
}}</a>.
            {{ course.total_modules }} modules.
            Instructor: {{ course.owner.get_full_name }}
        </p>
      {% endwith %}
    {% endfor %}
  </div>
{% endblock %}
```

Make sure that no template tag is split into multiple lines.

This is the template for listing the available courses. You create an HTML list to display all `Subject` objects and build a link to the `course_list_subject` URL for each of them. You also include the total number of courses for each subject and use the `pluralize` template filter to add a plural suffix to the word **course** when the number is different than `1`, to show *0 courses*, *1 course*, *2 courses*, etc. You add a `selected` HTML class to highlight the current subject if a subject is selected. You iterate over every `Course` object, displaying the total number of modules and the instructor's 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:

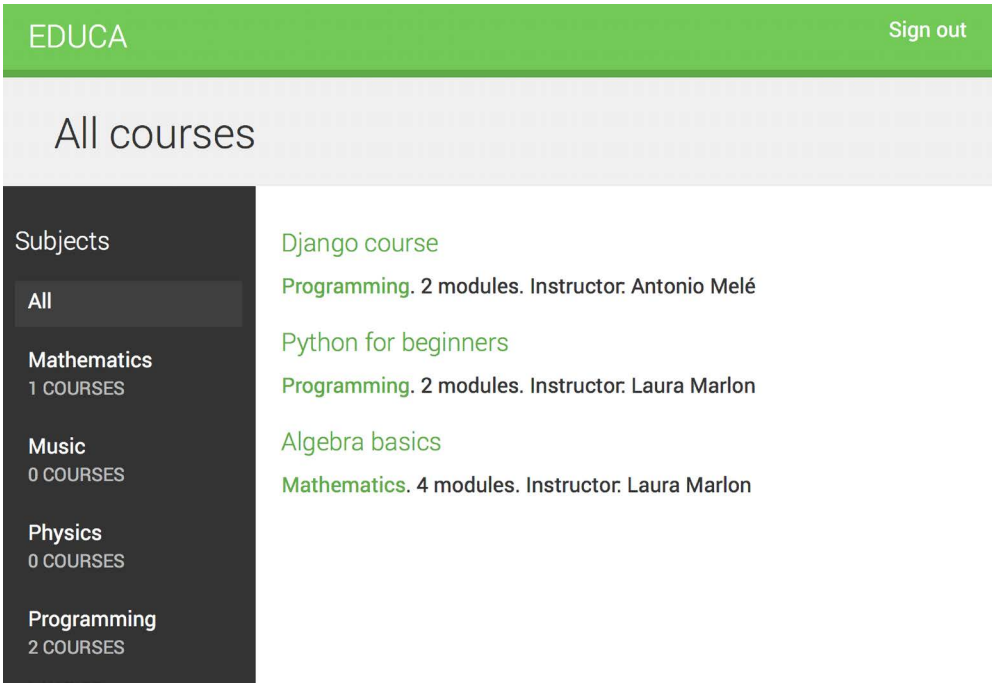


Figure 14.1: The course list page

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

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

Figure 14.2: The course overview page

You have created a public area for displaying courses. Next, you need to allow users to register as students and enroll on 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().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 your site. You 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`. You 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. You reverse the URL named `student_course_list`, which you are going to create in the *Accessing the course contents* section for listing the courses that students are enrolled on.

The `form_valid()` method is executed when valid form data has been posted. It has to return an HTTP response. You override this method to log the user in after they have successfully signed 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 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 a registration form like this:

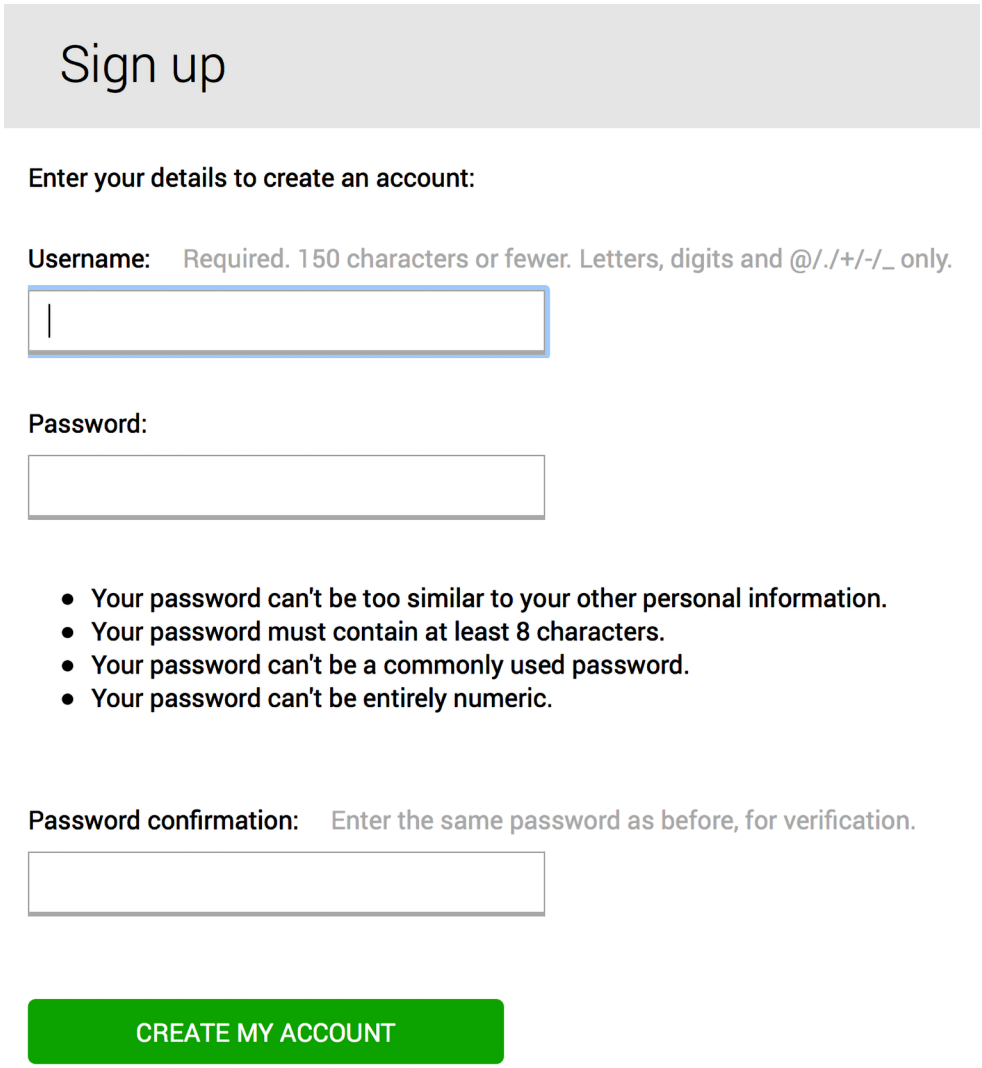


Figure 14.3: The student registration form

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 to after a successful registration. As mentioned, you will create this URL in the *Accessing the course contents* section.

Enrolling on courses

After users create an account, they should be able to enroll on courses. To store enrollments, you 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 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
```

You can now associate students with the courses on which they are enrolled. Let's create the functionality for students to enroll on 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)
```

You are going to use this form for students to enroll on courses. The `course` field is for the course on which the user will be enrolled; therefore, it's a `ModelChoiceField`. You use a `HiddenInput` widget because you are not going to show this field to the user. You 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().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 on 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 you handle a form submission. You 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, you add the current user to the students enrolled on the course.

The `get_success_url()` method returns the URL that the user will be redirected to if the form was successfully submitted. This method is equivalent to the `success_url` attribute. Then, you reverse the URL named `student_course_detail`.

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

You use the `get_context_data()` method to include the enrollment form in the context for rendering the templates. You 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 locate 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" value="Enroll now">
  </form>
{% else %}
  <a href="{% url "student_registration" %}" class="button">
    Register to enroll
  </a>
{% endif %}
```

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

Make sure that 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:

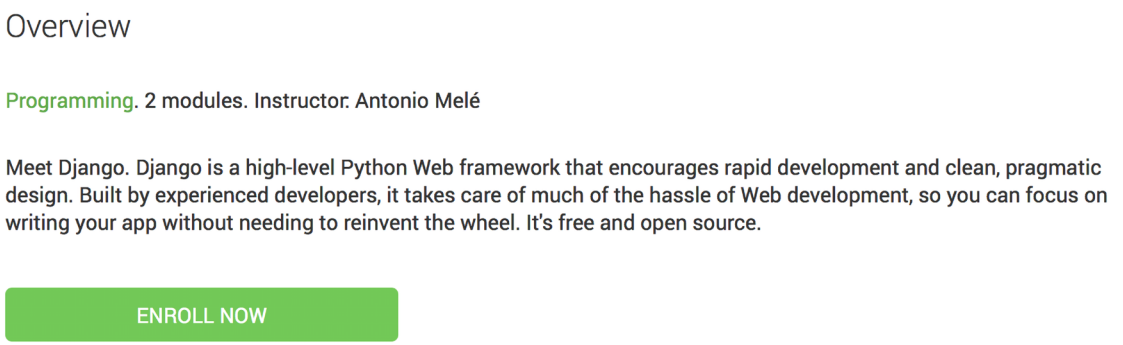


Figure 14.4: The course overview page, including an ENROLL NOW button

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

Accessing the course contents

You need a view for displaying the courses that students are enrolled on, 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().get_queryset()
        return qs.filter(students__in=[self.request.user])
```

This is the view to see courses that students are enrolled on. 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. You override the `get_queryset()` method to retrieve only the courses that a student is enrolled on; you filter the QuerySet by the student's `ManyToManyField` field to do so.

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

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

class StudentCourseDetailView(DetailView):
```

```
    model = Course
    template_name = 'students/course/detail.html'

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

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        # 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 the `StudentCourseDetailView` view. You override the `get_queryset()` method to limit the base QuerySet to courses on which the student is enrolled. You also override the `get_context_data()` method to set a course module in the context if the `module_id` URL parameter is given. Otherwise, you 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 on a course.
      </p>
    {% endfor %}
  </div>
{% endblock %}
```

This template displays the courses that the student is enrolled on. 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 student after a successful login if no `next` parameter is present in the request. After a successful login, a student will be redirected to the `student_course_list` URL to view the courses that they are enrolled on.

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 %}
```

Make sure no template tag is split across multiple lines. This is the template for enrolled students to access the contents of a course. First, you build an HTML list including all course modules and highlighting the current module. Then, you iterate over the current module contents and access each content item to display it using `{{ item.render }}`. You will add the `render()` method to the content models next. This method will take care of rendering the content properly.

You can now access `http://127.0.0.1:8000/students/register/`, register a new student account, and enroll on any course.

Rendering different types of content

To display the course contents, you need to render the different content types that you created: *text*, *image*, *video*, and *file*.

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

class ItemBase(models.Model):
    # ...
    def render(self):
        return render_to_string(
            f'courses/content/{self._meta.model_name}.html',
            {'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. You 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 }}
```

This is the template to render text content. The `linebreaks` template filter replaces line breaks in plain text with HTML line breaks.

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

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

This is the template to render files. You generate a link to download the file.

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

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

This is the template to render images.

You also have to create a template for rendering `Video` objects. You 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 their public URL.

Install the package with the following command:

```
pip install django-embed-video==1.4.4
```

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

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

You can find the `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" %}
```

This is the template to render videos.

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 on the course and redirected to the `student_course_detail` URL. *Figure 14.5* shows a sample course contents page:

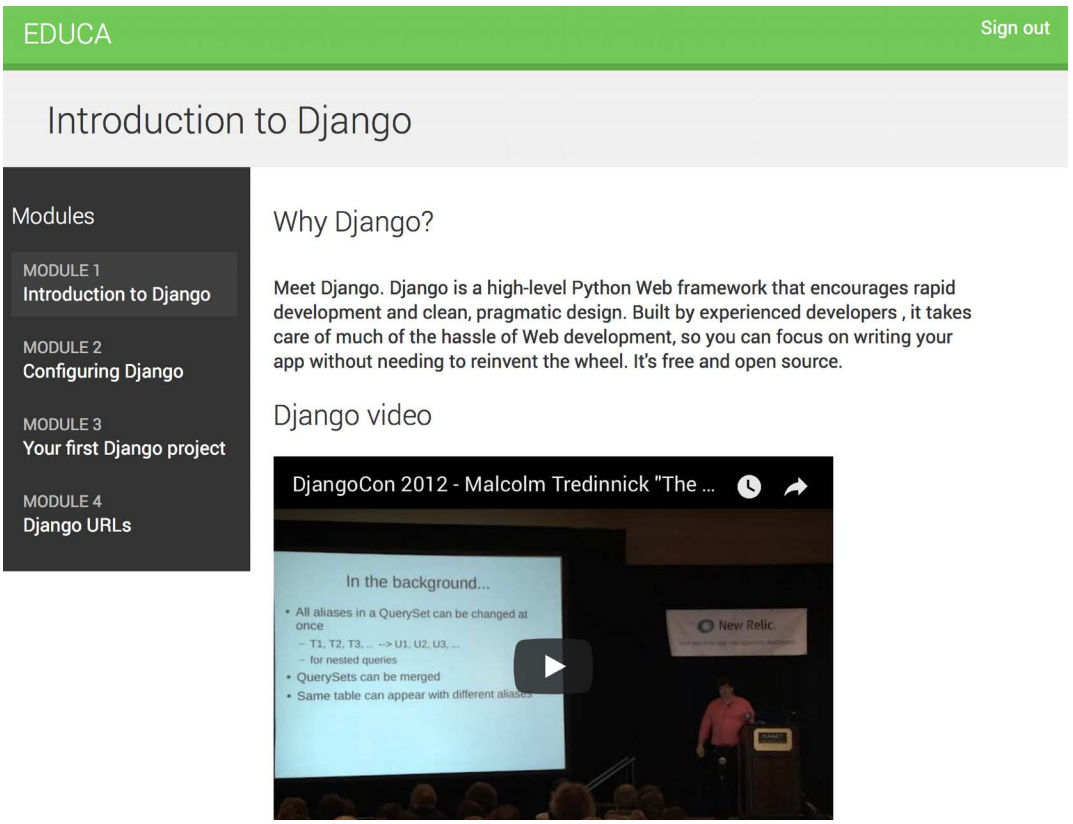


Figure 14.5: A course contents page

Great! You have created a common interface for rendering courses with different types of content.

Using the cache framework

Processing HTTP requests to your web application usually entails database access, data manipulation, and template rendering. It is much more expensive in terms of processing than just 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 expensive operations in the following requests that need to return the same data. 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, but you can specify the timeout when you cache data.

This is how you will usually use the cache framework when your application processes 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 database query or processing required to generate 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/4.1/topics/cache/`.

Available cache backends

Django comes with the following cache backends:

- `backends.memcached.PyMemcacheCache` or `backends.memcached.PyLibMCCache`: Memcached backends. Memcached is a fast and efficient memory-based cache server. The backend to use depends on the Memcached Python bindings you choose.
- `backends.redis.RedisCache`: A Redis cache backend. This backend has been added in Django 4.0.
- `backends.db.DatabaseCache`: Use the database as a cache system.
- `backends.filebased.FileBasedCache`: Use the file storage system. This serializes and stores each cache value as a separate file.
- `backends.locmem.LocMemCache`: A local memory cache backend. This is 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 or Redis backends.

Installing Memcached

Memcached is a popular high-performance, memory-based cache server. We are going to use Memcached and the `PyMemcacheCache` Memcached backend.

Installing the Memcached Docker image

Run the following command from the shell to pull the Memcached Docker image:

```
docker pull memcached
```

This will download the Memcached Docker image to your local machine. If you don't want to use Docker, you can also download Memcached from `https://memcached.org/downloads`.

Run the Memcached Docker container with the following command:

```
docker run -it --rm --name memcached -p 11211:11211 memcached -m 64
```

Memcached runs on port `11211` by default. The `-p` option is used to publish the `11211` port to the same host interface port. The `-m` option is used to limit the memory for the container to `64` MB. 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. If you want to run the command in detached mode (in the background of your terminal) you can use the `-d` option.

You can find more information about Memcached at `https://memcached.org`.

Installing the Memcached Python binding

After installing Memcached, you have to install a Memcached Python binding. We will install `pymemcache`, which is a fast, pure-Python Memcached client. Run the following command in the shell:

```
pip install pymemcache==3.5.2
```

You can read more information about the `pymemcache` library at `https://github.com/pinterest/pymemcache`.

Django 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 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. It is 300 seconds by default, which is 5 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 your 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.PyMemcacheCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

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

You have set up Memcached for your project. 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.
- **Template cache**: Allows you to cache template fragments.
- **Per-view cache**: Provides caching for individual views.
- **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 that are not calculated on a per-user basis.

Let's start by learning how to use the low-level cache API in your Python code.

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 Django shell with the following command:

```
python manage.py shell
```

Execute the following code:

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

You 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 you 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'
```

You 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 has 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('my_subjects', subjects)
```

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

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

You are going to cache some queries in your 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`, find the following lines:

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

Replace the lines 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, you 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), you perform the query to retrieve all `Subject` objects and their number of courses, and you cache the result using `cache.set()`.

Checking cache requests with Django Debug Toolbar

Let's add Django Debug Toolbar to the project to check the cache queries. You learned how to use Django Debug Toolbar in *Chapter 7, Tracking User Actions*.

First install Django Debug Toolbar with the following command:

```
pip install django-debug-toolbar==3.6.0
```

Edit the `settings.py` file of your project and add `debug_toolbar` to the `INSTALLED_APPS` setting as follows. The new line is highlighted in bold:

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

In the same file, add the following line highlighted in bold to the `MIDDLEWARE` setting:

```
MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
```

```
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Remember that `DebugToolbarMiddleware` has to be placed before any other middleware, except for middleware that encodes the response's content, such as `GZipMiddleware`, which, if present, should come first.

Add the following lines at the end of the `settings.py` file:

```
INTERNAL_IPS = [
    '127.0.0.1',
]
```

Django Debug Toolbar will only display if your IP address matches an entry in the `INTERNAL_IPS` setting.

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

```
path('__debug__/', include('debug_toolbar.urls')),]
```

Run the development server and open `http://127.0.0.1:8000/` in your browser.

You should now see Django Debug Toolbar on the right side of the page. Click on **Cache** in the sidebar menu. You will see the following panel:

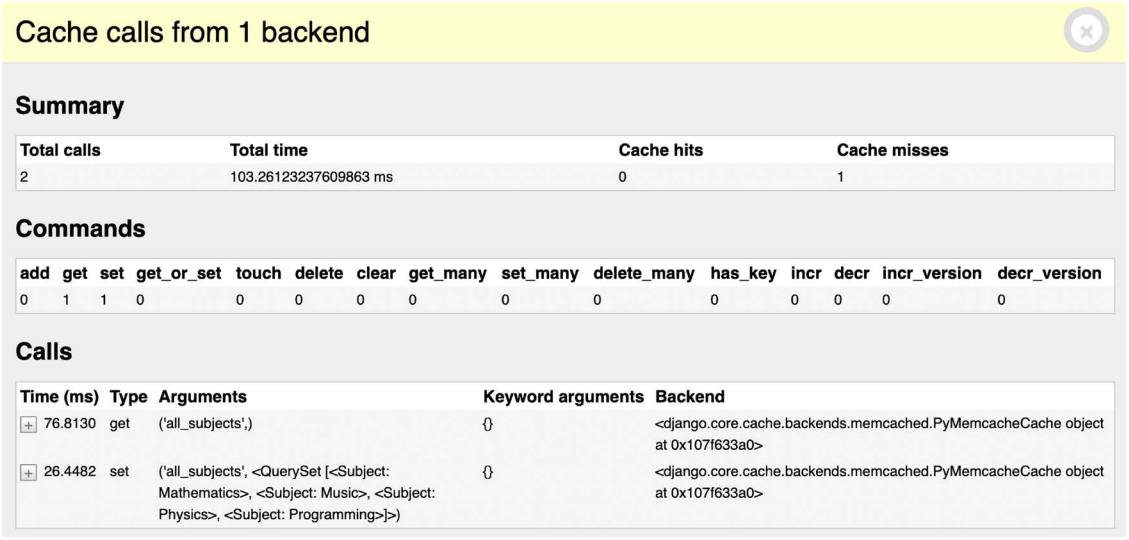


Figure 14.6: The Cache panel of Django Debug Toolbar including cache requests for CourseListView on a cache miss

Under **Total calls** you should see `2`. The first time the `CourseListView` view is executed there are two cache requests. Under **Commands** you will see that the `get` command has been executed once, and that the `set` command has been executed once as well. The `get` command corresponds to the call that retrieves the `all_subjects` cache key. This is the first call displayed under **Calls**. The first time the view is executed a cache miss occurs because no data is cached yet. That's why there is `1` under **Cache misses**. Then, the `set` command is used to store the results of the subjects QuerySet in the cache using the `all_subjects` cache key. This is the second call displayed under **Calls**.

In the **SQL** menu item of Django Debug Toolbar, you will see the total number of SQL queries executed in this request. This includes the query to retrieve all subjects that are then stored in the cache:

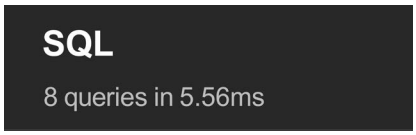


Figure 14.7: SQL queries executed for CourseListView on a cache miss

Reload the page in the browser and click on **Cache** in the sidebar menu:

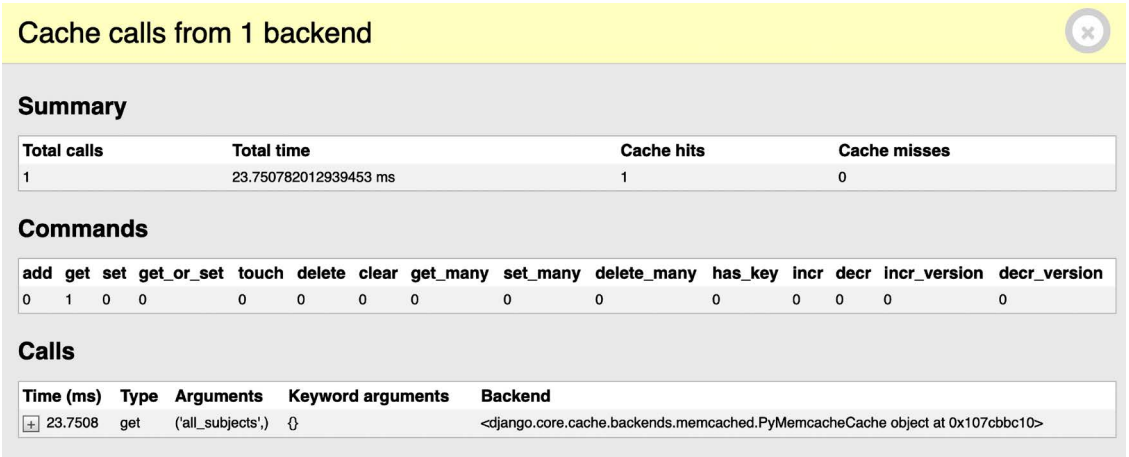


Figure 14.8: The Cache panel of Django Debug Toolbar, including cache requests for CourseListView view on a cache hit

Now, there is only a single cache request. Under **Total calls** you should see `1`. And under **Commands** you can see that the cache request corresponds to a `get` command. In this case there is a cache hit (see **Cache hits**) instead of a cache miss because the data has been found in the cache. Under **Calls** you can see the `get` request to retrieve the `all_subjects` cache key.

Check the **SQL** menu item of the debug toolbar. You should see that there is one less SQL query in this request. You are saving one SQL query because the view finds the data in the cache and doesn't need to retrieve it from the database:

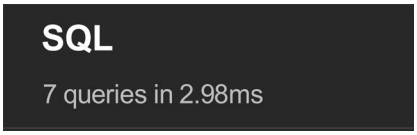


Figure 14.9: SQL queries executed for CourseListView on a cache hit

In this example, for a single request, it takes more time to retrieve the item from the cache than the time saved on the additional SQL query. However, when you have many users accessing your site, you will find significant time reductions by retrieving the data from the cache instead of hitting the database, and you will be able to serve the site to more concurrent users.

Successive requests to the same URL will retrieve the data from the cache. Since we didn't specify a timeout when caching data with `cache.set('all_subjects', subjects)` in the `CourseListView` view, the default timeout will be used (300 seconds by default, which is 5 minutes). When the timeout is reached, the next request to the URL will generate a cache miss, the QuerySet will be executed, and data will be cached for another 5 minutes. You can define a different default timeout in the `TIMEOUT` element of the `CACHES` setting.

Caching based on dynamic data

Often, you will want to cache something that is based on dynamic data. In these cases, you have to build dynamic keys that contain all the 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 = f'subject_{subject.id}_courses'
            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, you also cache both all courses and courses filtered by subject. You use the `all_courses` cache key for storing all courses if no subject is given. If there is a subject, you build the key dynamically with `f'subject_{subject.id}_courses'`.

It's important to note that you can't use a cached QuerySet to build other QuerySets, since what you cached are actually the results of the QuerySet. So you can't 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 %}` template 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, find 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 %}
```

You cache this template fragment using the name `module_contents` and pass the current `Module` object to it. Thus, you 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 your 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 complete content returned by 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 = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Remember that middleware is 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 middleware is executed in reverse order. `FetchFromCacheMiddleware` is placed after `CommonMiddleware` intentionally because it needs to access request data set by the latter.

Next, 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, you use the default cache for your cache middleware and set the global cache timeout to 15 minutes. You also specify a prefix for all cache keys to avoid collisions in case you use the same Memcached backend for multiple projects. Your site will now cache and return cached content for all `GET` requests.

You can access the different pages and check the cache requests using Django Debug Toolbar. The per-site cache is not viable for many sites because it affects all views, even the ones that you might not want to cache, like management views where you want data to be returned from the database to reflect the latest changes.

In this project, the best approach is to cache the templates or views that are used to display course contents to students, while keeping the content management views for instructors without any cache.

Let's deactivate the per-site cache. Edit the `settings.py` file of your project and comment out the `UpdateCacheMiddleware` and `FetchFromCacheMiddleware` classes in the `MIDDLEWARE` setting, as follows:

```
MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    # 'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.cache.FetchFromCacheMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

You have seen an overview of the different methods provided by Django to cache data. You should always define your cache strategy wisely, taking into account expensive QuerySets or calculations, data that won't change frequently, and data that will be accessed concurrently by many users.

Using the Redis cache backend

Django 4.0 introduced a Redis cache backend. Let's change the settings to use Redis instead of Memcached as the cache backend for the project. Remember that you already used Redis in *Chapter 7*, *Tracking User Actions*, and in *Chapter 10*, *Extending Your Shop*.

Install `redis-py` in your environment using the following command:

```
pip install redis==4.3.4
```

Then, edit the `settings.py` file of the `educa` project and modify the `CACHES` setting, as follows:

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

The project will now use the `RedisCache` cache backend. The location is defined in the format `redis://[host]:[port]`. You use `127.0.0.1` to point to the local host and `6379`, which is the default port for Redis.

Initialize the Redis Docker container using the following command:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

If you want to run the command in the background (in detached mode) you can use the `-d` option.

Run the development server and open `http://127.0.0.1:8000/` in your browser. Check the cache requests in the **Cache** panel of Django Debug Toolbar. You are now using Redis as your project's cache backend instead of Memcached.

Monitoring Redis with Django Redisboard

You can monitor your Redis server using Django Redisboard. Django Redisboard adds Redis statistics to the Django administration site. You can find more information about Django Redisboard at `https://github.com/ionelmc/django-redisboard`.

Install `django-redisboard` in your environment using the following command:

```
pip install django-redisboard==8.3.0
```

Install the `attrs` Python library used by `django-redisboard` in your environment with the following command:

```
pip install attrs
```

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

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

Run the following command from your project's directory to run the Django Redisboard migrations:

```
python manage.py migrate redisboard
```

Run the development server and open `http://127.0.0.1:8000/admin/redisboard/redisserver/add/` in your browser to add a Redis server to monitor. Under the **Label**, enter `redis`, and under **URL**, enter `redis://localhost:6379/0`, as in *Figure 14.10*:

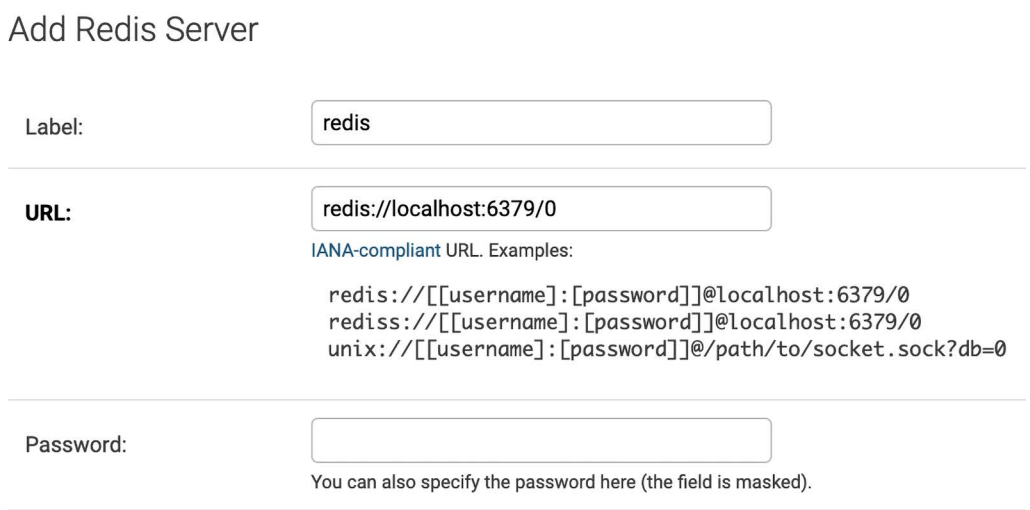


Figure 14.10: The form to add a Redis server for Django Redisboard in the administration site

We will monitor the Redis instance running on our local host, which runs on port `6379` and uses the Redis database numbered `0`. Click on **SAVE**. The information will be saved to the database, and you will be able to see the Redis configuration and metrics on the Django administration site:

Figure 14.11: The Redis monitoring of Django Redisboard on the administration site

Congratulations! You have successfully implemented caching for your project.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter14`
- `django-embed-video` documentation – `https://django-embed-video.readthedocs.io/en/latest/`
- Django's cache framework documentation – `https://docs.djangoproject.com/en/4.1/topics/cache/`
- Memcached downloads – `https://memcached.org/downloads`
- Memcached official website – `https://memcached.org`
- `Pymemcache`'s source code – `https://github.com/pinterest/pymemcache`
- Django Redisboard's source code – `https://github.com/ionelmc/django-redisboard`

Summary

In this chapter, you implemented the public views for the course catalog. You built a system for students to register and enroll on courses. You also created the functionality to render different types of content for the course modules. Finally, you learned how to use the Django cache framework and you used the Memcached and Redis cache backends for your project.

In the next chapter, you will build a RESTful API for your project using Django REST framework and consume it using the Python Requests library.

15

Building an API

In the previous chapter, you built a system for student registration and enrollment on courses. You created views to display course contents and learned how to use Django's cache framework.

In this chapter, you will create a RESTful API for your e-learning platform. An API allows you to build a common core that can be used on multiple platforms like websites, mobile applications, plugins, and so on. For example, you can create an API to be consumed by a mobile application for your e-learning platform. If you provide an API to third parties, they will be able to consume information and operate with your application programmatically. An API allows developers to automate actions on your platform and integrate your service with other applications or online services. You will build a fully featured API for your e-learning platform.

In this chapter, you will:

- Install Django REST framework
- Create serializers for your models
- Build a RESTful API
- Create nested serializers
- Build custom API views
- Handle API authentication
- Add permissions to API views
- Create a custom permission
- Implement `ViewSets` and routers
- Use the Requests library to consume the API

Let's start with the setup of your API.

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter15`.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `pip install -r requirements.txt`.

Building a RESTful API

When building an API, there are several ways you can structure its endpoints and actions, 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. You will build a RESTful API with JSON serialization for your project. Your API will provide the following functionality:

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

You 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 is Django REST framework.

Installing Django REST framework

Django REST framework allows you to easily build RESTful APIs for your project. You can find all the 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==3.13.1
```

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. You 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, you need to specify how your data will be serialized. Output data has to be serialized in a specific format, and input data will be deserialized 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 your first serializer. Create the following file structure inside the `courses` application directory:

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

You 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 courses.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 the 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, you 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 deserialize 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 by inspecting 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, the URL `http://127.0.0.1:8000/api/data.json` might be an endpoint that triggers 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"}'
```

You 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.

Next, you are going to learn how to build API views and use serializers in views.

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. They provide the functionality to retrieve, create, update, or delete model objects. You can see all the 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 courses.models import Subject
from courses.api.serializers import SubjectSerializer

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

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

In this code, you are using the generic `ListAPIView` and `RetrieveAPIView` views of REST framework. You 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 your 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')),
]
```

Our initial API endpoints are now ready to be used.

Consuming the API

You use the `api` namespace for your API URLs. Ensure that your server is running with the following command:

```
python manage.py runserver
```

We are going to use `curl` to consume the API. `curl` is a command-line tool that allows you to transfer data to and from a server. If you are using Linux, macOS, or Windows 10/11, `curl` is very likely included in your system. However, you can download `curl` from `https://curl.se/download.html`.

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"
    }
]
```

To obtain a more readable, well-indented JSON response, you can use `curl` with the `json_pp` utility, as follows:

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

The HTTP response contains a list of `Subject` objects in JSON format.

Instead of `curl`, you can also use any other tool to send custom HTTP requests, including 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:

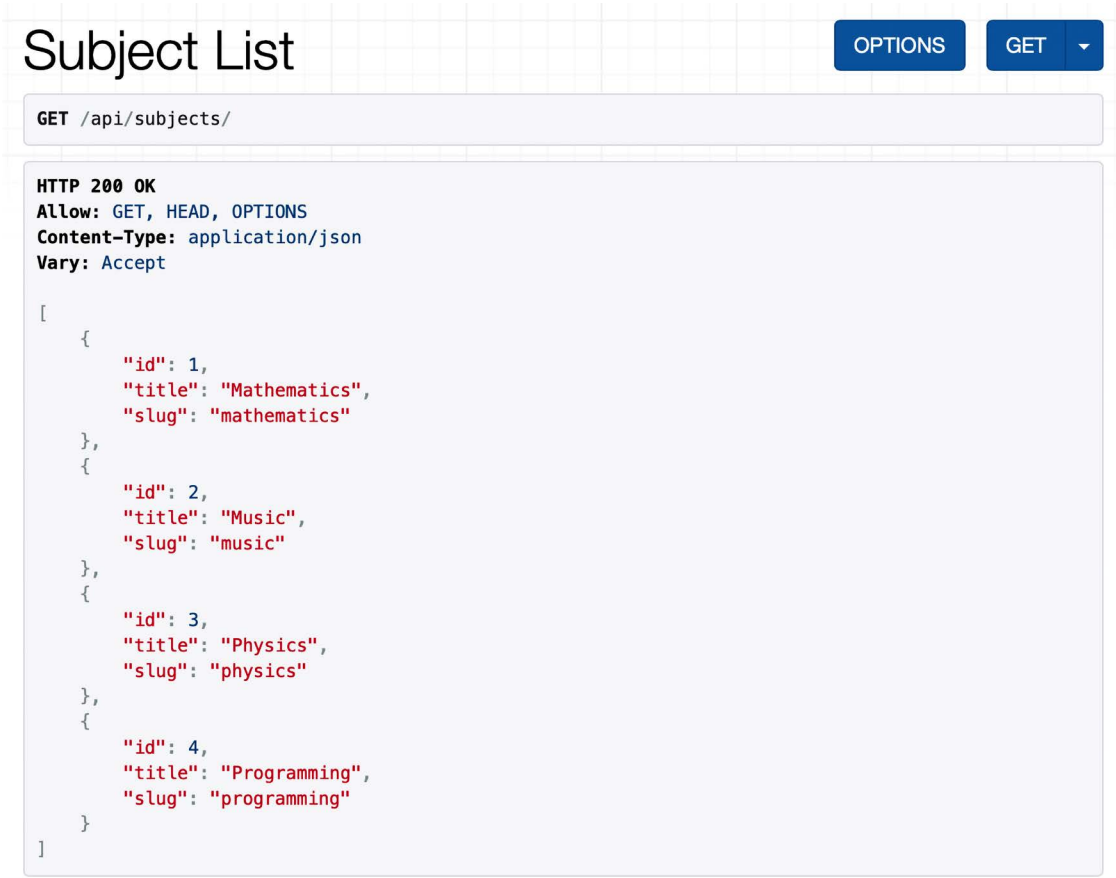


Figure 15.1: The subject list page in the REST framework browsable API

This HTML interface is provided by the `BrowsableAPIRenderer` renderer. It displays the result headers and content, and it 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.

Figure 15.2: The subject detail page in the REST framework browsable API

This is the response for the `SubjectDetailView`. Next, we are going to dig deeper into model serializers.

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 highlighted in bold:

```
from courses.models import Subject, 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 and execute the following command:

```
python manage.py shell
```

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 that you 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]
```

You want to include more information about each module, so you 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 courses.models import Subject, Course, Module

class ModuleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Module
        fields = ['order', 'title', 'description']


class CourseSerializer(serializers.ModelSerializer):
    modules = ModuleSerializer(many=True, read_only=True)

    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug',
                  'overview', 'created', 'owner',
                  'modules']
```

In the new code, you define `ModuleSerializer` to provide serialization for the `Module` model. Then, you add a `modules` attribute to `CourseSerializer` to nest the `ModuleSerializer` serializer. You set `many=True` to indicate that you 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.org/api-guide/serializers/`.

Generic API views are very useful to build REST APIs based on your models and serializers. However, you might also need to implement your own views with custom logic. Let's learn how to create a custom API view.

Building custom API views

REST framework provides an `APIView` class that builds API functionality on top of Django's `View` class. The `APIView` class differs from `View` by 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.

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

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import generics
from courses.models import Subject, Course
from courses.api.serializers import SubjectSerializer

# ...

class CourseEnrollView(APIView):
```

```
    def post(self, request, pk, format=None):
        course = get_object_or_404(Course, pk=pk)
        course.students.add(request.user)
        return Response({'enrolled': True})
```

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

1. You create a custom view that subclasses `APIView`.
2. You define a `post()` method for `POST` actions. No other HTTP method will be allowed for this view.
3. You expect a `pk` URL parameter containing the ID of a course. You retrieve the course by the given `pk` parameter and raise a `404` exception if it's not found.
4. You 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, you could now perform a `POST` request to enroll the current user on a course. However, you 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 uses Django's session backend for authentication. This backend is useful for performing 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 your 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 whether 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]
    # ...
```

You include the `IsAuthenticated` permission. This will prevent anonymous users from accessing the view. Now, you can perform a `POST` request to your 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."}
```

You got a `401` HTTP code as expected since you are not authenticated. Let's use basic authentication with one of your 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.

Next, you are going to learn a different way to build common views by using `ViewSets`.

Creating ViewSets 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 `ViewSets`, you can avoid repeating logic for multiple views. `ViewSets` include actions for the following standard operations:

- Create operation: `create()`
- Retrieve operation: `list()` and `retrieve()`

- Update operation: `update()` and `partial_update()`
- Delete operation: `destroy()`

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

```
# ...
from rest_framework import viewsets
from courses.api.serializers import SubjectSerializer,
     CourseSerializer

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

You subclass `ReadOnlyModelViewSet`, which provides the read-only actions `list()` and `retrieve()` to both list objects, or retrieves a single object.

Edit the `api/urls.py` file and create a router for your `ViewSet`, 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)),
]
```

You create a `DefaultRouter` object and register your `ViewSet` with the `courses` prefix. The router takes charge of generating URLs automatically for your `ViewSet`.

Open `http://127.0.0.1:8000/api/` in your browser. You will see that the router lists all `ViewSets` in its base URL, as shown in *Figure 15.3*:

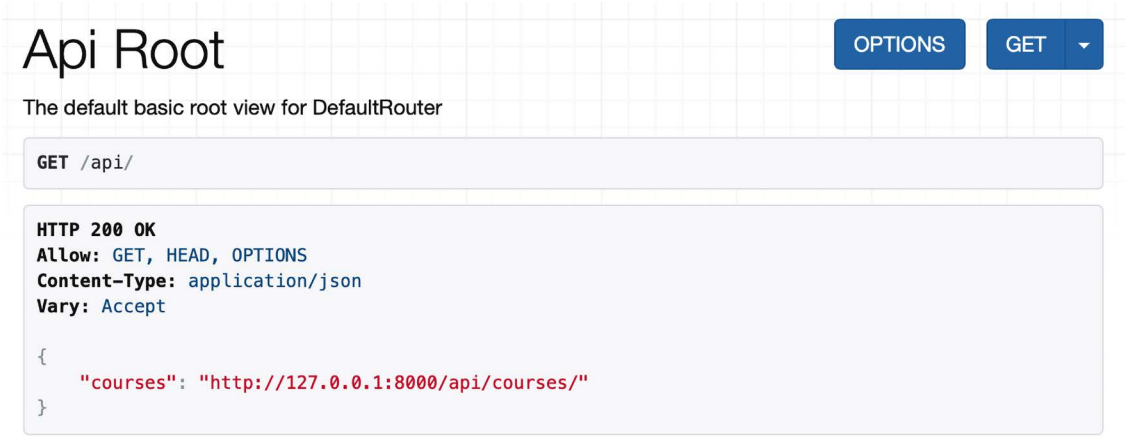


Figure 15.3: The API root page of the REST framework browsable API

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

You can learn more about `ViewSets` 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 ViewSets

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

```
# ...
from rest_framework.decorators import action

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

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

In the preceding code, you add a custom `enroll()` method that represents an additional action for this `ViewSet`. The preceding code is as follows:

1. You use the `action` decorator of the framework with the parameter `detail=True` to specify that this is an action to be performed on a single object.
2. The decorator allows you to add custom attributes for the action. You specify that only the `post()` method is allowed for this view and set the authentication and permission classes.
3. You use `self.get_object()` to retrieve the `Course` object.
4. You 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 or comment out the following URL, since you 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 or comment out the `CourseEnrollView` class.

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

After students are enrolled in a course, they need to access the course's content. Next, you are going to learn how to ensure only students that enrolled can access the course.

Creating custom permissions

You want students to be able to access the contents of the courses they are enrolled on. Only students enrolled on a course should be able to access its contents. The best way to do this is with a custom permission class. REST Framework 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()
```

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

Serializing course contents

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

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

```
from courses.models import Subject, Course, Module, Content

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

class ContentSerializer(serializers.ModelSerializer):
    item = ItemRelatedField(read_only=True)
    class Meta:
        model = Content
        fields = ['order', 'item']
```

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

You need an alternative 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 includes the course contents. Edit the `api/views.py` file and add the following method to the `CourseViewSet` class:

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

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    # ...
    @action(detail=True,
            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:

1. You use the `action` decorator with the parameter `detail=True` to specify an action that is performed on a single object.
2. You specify that only the `GET` method is allowed for this action.
3. You use the new `CourseWithContentsSerializer` serializer class that includes rendered course contents.
4. You use both `IsAuthenticated` and your custom `IsEnrolled` permissions. By doing so, you make sure that only users enrolled in the course are able to access its contents.
5. You 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` class. 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/`.

Consuming the RESTful API

Now that you have implemented an API, you can consume it in a programmatic manner from other applications. You can interact with the API using the JavaScript Fetch API in the frontend of your application, in a similar fashion to the functionalities you built in *Chapter 6*, *Sharing Content on Your Website*. You can also consume the API from applications built with Python or any other programming language.

You are going to create a simple Python application that uses the RESTful API to retrieve all available courses and then enroll a student in all of them. You will learn how to authenticate against the API using HTTP basic authentication and perform `GET` and `POST` requests.

We will use the Python Requests library to consume the API. We used Requests in *Chapter 6*, *Sharing Content on Your Website* to retrieve images by their URL. Requests abstracts the complexity of dealing with HTTP requests and provides a very simple interface to consume HTTP services. You can find the documentation for the Requests library at `https://requests.readthedocs.io/en/master/`.

Open the shell and install the Requests library with the following command:

```
pip install requests==2.28.1
```

Create a new directory next to the `educa` project directory and name it `api_examples`. Create a new file inside the `api_examples/` directory and name it `enroll_all.py`. The file structure should now look like this:

```
api_examples/
    enroll_all.py
educa/
    ...
```

Edit the `enroll_all.py` file and add the following code to it:

```
import requests

base_url = 'http://127.0.0.1:8000/api/'

# retrieve all courses
r = requests.get(f'{base_url}courses/')
courses = r.json()

available_courses = ', '.join([course['title'] for course in courses])
print(f'Available courses: {available_courses}')
```

In this code, you perform the following actions:

1. You import the Requests library and define the base URL for the API.
2. You use `requests.get()` to retrieve data from the API by sending a `GET` request to the URL `http://127.0.0.1:8000/api/courses/`. This API endpoint is publicly accessible, so it does not require any authentication.
3. You use the `json()` method of the response object to decode the JSON data returned by the API.
4. You print the title attribute of each course.

Start the development server from the `educa` project directory with the following command:

```
python manage.py runserver
```

In another shell, run the following command from the `api_examples/` directory:

```
python enroll_all.py
```

You will see output with a list of all course titles, like this:

```
Available courses: Introduction to Django, Python for beginners, Algebra basics
```

This is your first automated call to your API.

Edit the `enroll_all.py` file and change it to make it look like this:

```
import requests

username = ''
password = ''
base_url = 'http://127.0.0.1:8000/api/'

# retrieve all courses
r = requests.get(f'{base_url}courses/')
courses = r.json()
available_courses = ', '.join([course['title'] for course in courses])
print(f'Available courses: {available_courses}')

for course in courses:
  course_id = course['id']
  course_title = course['title']
  r = requests.post(f'{base_url}courses/{course_id}/enroll/',
                    auth=(username, password))
  if r.status_code == 200:
    # successful request
    print(f'Successfully enrolled in {course_title}')
```

Replace the values for the `username` and `password` variables with the credentials of an existing user.

With the new code, you perform the following actions:

1. You define the username and password of the student you want to enroll on courses.
2. You iterate over the available courses retrieved from the API.
3. You store the course ID attribute in the `course_id` variable and the title attribute in the `course_title` variable.
4. You use `requests.post()` to send a `POST` request to the URL `http://127.0.0.1:8000/api/courses/[id]/enroll/` for each course. This URL corresponds to the `CourseEnrollView` API view, which allows you to enroll a user on a course. You build the URL for each course using the `course_id` variable. The `CourseEnrollView` view requires authentication. It uses the `IsAuthenticated` permission and the `BasicAuthentication` authentication class. The Requests library supports HTTP basic authentication out of the box. You use the `auth` parameter to pass a tuple with the username and password to authenticate the user using HTTP basic authentication.
5. If the status code of the response is `200 OK`, you print a message to indicate that the user has been successfully enrolled on the course.

You can use different kinds of authentication with Requests. You can find more information on authentication with Requests at `https://requests.readthedocs.io/en/master/user/authentication/`.

Run the following command from the `api_examples/` directory:

```
python enroll_all.py
```

You will now see output like this:

```
Available courses: Introduction to Django, Python for beginners, Algebra basics
Successfully enrolled in Introduction to Django
Successfully enrolled in Python for beginners
Successfully enrolled in Algebra basics
```

Great! You have successfully enrolled the user on all available courses using the API. You will see a `Successfully enrolled` message for each course on the platform. As you can see, it's very easy to consume the API from any other application. You can effortlessly build other functionalities based on the API and let others integrate your API into their applications.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter15`
- REST framework website – `https://www.django-rest-framework.org/`
- REST framework settings – `https://www.django-rest-framework.org/api-guide/settings/`
- REST framework renderers – `https://www.django-rest-framework.org/api-guide/renderers/`
- REST framework parsers – `https://www.django-rest-framework.org/api-guide/parsers/`
- REST framework generic mixins and views – `https://www.django-rest-framework.org/api-guide/generic-views/`
- Download `curl` – `https://curl.se/download.html`
- Postman API platform – `https://www.getpostman.com/`
- REST framework serializers – `https://www.django-rest-framework.org/api-guide/serializers/`
- HTTP basic authentication – `https://en.wikipedia.org/wiki/Basic_access_authentication`
- REST framework authentication – `https://www.django-rest-framework.org/api-guide/authentication/`
- REST framework permissions – `https://www.django-rest-framework.org/api-guide/permissions/`
- REST framework `ViewSets` – `https://www.django-rest-framework.org/api-guide/viewsets/`
- REST framework routers – `https://www.django-rest-framework.org/api-guide/routers/`
- Python Requests library documentation – `https://requests.readthedocs.io/en/master/`
- Authentication with the Requests library – `https://requests.readthedocs.io/en/master/user/authentication/`

Summary

In this chapter, you learned how to use Django REST framework to build a RESTful API for your project. You created serializers and views for models, and you built custom API views. You also added authentication to your API and restricted access to API views using permissions. Next, you discovered how to create custom permissions, and you implemented `ViewSets` and routers. Finally, you used the Requests library to consume the API from an external Python script.

The next chapter will teach you how to build a chat server using Django Channels. You will implement asynchronous communication using WebSockets and you will use Redis to set up a channel layer.

16

Building a Chat Server

In the previous chapter, you created a RESTful API for your project. In this chapter, you will build a chat server for students using Django Channels. Students will be able to access a different chat room for each course they are enrolled on. To create the chat server, you will learn how to serve your Django project through **Asynchronous Server Gateway Interface** (**ASGI**), and you will implement asynchronous communication.

In this chapter, you will:

- Add Channels to your project
- Build a WebSocket consumer and appropriate routing
- Implement a WebSocket client
- Enable a channel layer with Redis
- Make your consumer fully asynchronous

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter16`.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `pip install -r requirements.txt`.

Creating a chat application

You are going to implement a chat server to provide students with a chat room for each course. Students enrolled on a course will be able to access the course chat room and exchange messages in real time. You will use Channels to build this functionality. Channels is a Django application that extends Django to handle protocols that require long-running connections, such as WebSockets, chatbots, or MQTT (a lightweight publish/subscribe message transport commonly used in **Internet of Things** (**IoT**) projects).

Using Channels, you can easily implement real-time or asynchronous functionalities into your project in addition to your standard HTTP synchronous views. You will start by adding a new application to your project. The new application will contain the logic for the chat server.

You can the documentation for Django Channels at `https://channels.readthedocs.io/`.

Let's start implementing the chat server. Run the following command from the project `educa` directory to create the new application file structure:

```
django-admin startapp chat
```

Edit the `settings.py` file of the `educa` project and activate the `chat` application in your project by editing the `INSTALLED_APPS` setting, as follows:

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

The new `chat` application is now active in your project.

Implementing the chat room view

You will provide students with a different chat room for each course. You need to create a view for students to join the chat room of a given course. Only students who are enrolled on a course will be able to access the course chat room.

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

```
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseForbidden
from django.contrib.auth.decorators import login_required

@login_required
def course_chat_room(request, course_id):
    try:
        # retrieve course with given id joined by the current user
        course = request.user.courses_joined.get(id=course_id)
    except:
        # user is not a student of the course or course does not exist
        return HttpResponseForbidden()
    return render(request, 'chat/room.html', {'course': course})
```

This is the `course_chat_room` view. In this view, you use the `@login_required` decorator to prevent any non-authenticated user from accessing the view. The view receives a required `course_id` parameter that is used to retrieve the course with the given `id`.

You access the courses that the user is enrolled on through the relationship `courses_joined` and you retrieve the course with the given `id` from that subset of courses. If the course with the given `id` does not exist or the user is not enrolled on it, you return an `HttpResponseForbidden` response, which translates to an HTTP response with status `403`.

If the course with the given `id` exists and the user is enrolled on it, you render the `chat/room.html` template, passing the `course` object to the template context.

You need to add a URL pattern for this view. Create a new file inside the `chat` application directory and name it `urls.py`. Add the following code to it:

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

app_name = 'chat'

urlpatterns = [
    path('room/<int:course_id>/', views.course_chat_room,
         name='course_chat_room'),
]
```

This is the initial URL patterns file for the `chat` application. You define the `course_chat_room` URL pattern, including the `course_id` parameter with the `int` prefix, as you only expect an integer value here.

Include the new URL patterns of the `chat` application in the main URL patterns of the project. Edit the main `urls.py` file of the `educa` project and add the following line to it:

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

URL patterns for the `chat` application are added to the project under the `chat/` path.

You need to create a template for the `course_chat_room` view. This template will contain an area to visualize the messages that are exchanged in the chat, and a text input with a submit button to send text messages to the chat.

Create the following file structure within the `chat` application directory:

```
templates/
    chat/
        room.html
```

Edit the `chat/room.html` template and add the following code to it:

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

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

{% block content %}
  <div id="chat">
```

```
  </div>
  <div id="chat-input">
    <input id="chat-message-input" type="text">
    <input id="chat-message-submit" type="submit" value="Send">
  </div>
{% endblock %}

{% block include_js %}
{% endblock %}

{% block domready %}
{% endblock %}
```

This is the template for the course chat room. In this template, you extend the `base.html` template of your project and fill its `content` block. In the template, you define a `<div>` HTML element with the `chat` ID that you will use to display the chat messages sent by the user and by other students. You also define a second `<div>` element with a `text` input and a submit button that will allow the user to send messages. You add the `include_js` and `domready` blocks defined in the `base.html` template, which you are going to implement later, to establish a connection with a WebSocket and send or receive messages.

Run the development server and open `http://127.0.0.1:8000/chat/room/1/` in your browser, replacing `1` with the `id` of an existing course in the database. Access the chat room with a logged-in user who is enrolled on the course. You will see the following screen:

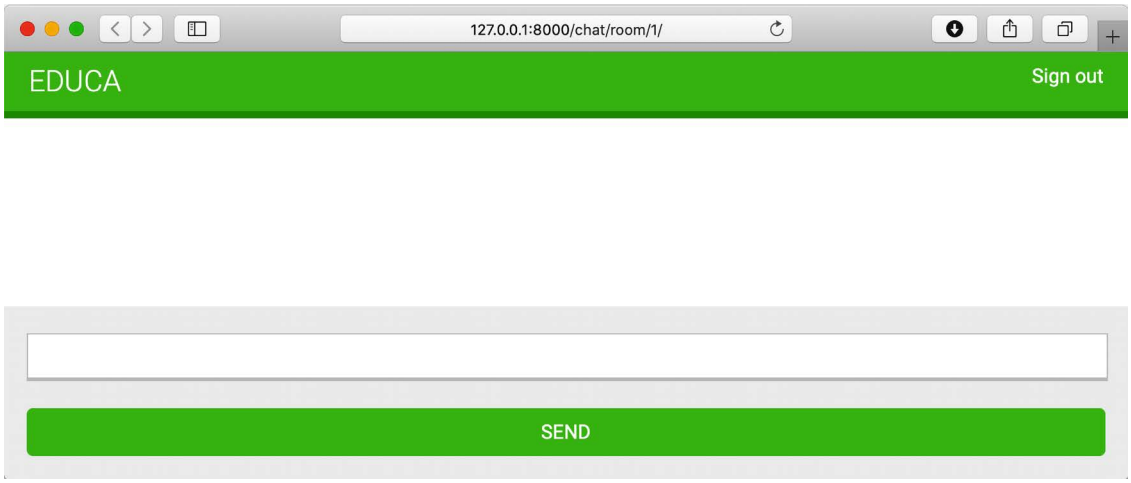


Figure 16.1: The course chat room page

This is the course chat room screen that students will use to discuss topics within a course.

Real-time Django with Channels

You are building a chat server to provide students with a chat room for each course. Students enrolled on a course will be able to access the course chat room and exchange messages. This functionality requires real-time communication between the server and the client. The client should be able to connect to the chat and send or receive data at any time. There are several ways you could implement this feature, using AJAX polling or long polling in combination with storing the messages in your database or Redis. However, there is no efficient way to implement a chat server using a standard synchronous web application. You are going to build a chat server using asynchronous communication through ASGI.

Asynchronous applications using ASGI

Django is usually deployed using **Web Server Gateway Interface** (**WSGI**), which is the standard interface for Python applications to handle HTTP requests. However, to work with asynchronous applications, you need to use another interface called ASGI, which can handle WebSocket requests as well. ASGI is the emerging Python standard for asynchronous web servers and applications.

You can find an introduction to ASGI at `https://asgi.readthedocs.io/en/latest/introduction.html`.

Django comes with support for running asynchronous Python through ASGI. Writing asynchronous views is supported since Django 3.1 and Django 4.1 introduces asynchronous handlers for class-based views. Channels builds upon the native ASGI support available in Django and provides additional functionalities to handle protocols that require long-running connections, such as WebSockets, IoT protocols, and chat protocols.

WebSockets provide full-duplex communication by establishing a persistent, open, bidirectional **Transmission Control Protocol** (**TCP**) connection between servers and clients. You are going to use WebSockets to implement your chat server.

You can find more information about deploying Django with ASGI at `https://docs.djangoproject.com/en/4.1/howto/deployment/asgi/`.

You can find more information about Django's support for writing asynchronous views at `https://docs.djangoproject.com/en/4.1/topics/async/` and Django's support for asynchronous class-based views at `https://docs.djangoproject.com/en/4.1/topics/class-based-views/#async-class-based-views`.

The request/response cycle using Channels

It's important to understand the differences in a request cycle between a standard synchronous request cycle and a Channels implementation. The following schema shows the request cycle of a synchronous Django setup:

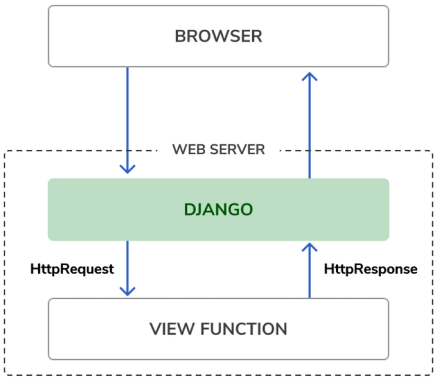


Figure 16.2: The Django request/response cycle

When an HTTP request is sent by the browser to the web server, Django handles the request and passes the `HttpRequest` object to the corresponding view. The view processes the request and returns an `HttpResponse` object that is sent back to the browser as an HTTP response. There is no mechanism to maintain an open connection or send data to the browser without an associated HTTP request.

The following schema shows the request cycle of a Django project using Channels with WebSockets:

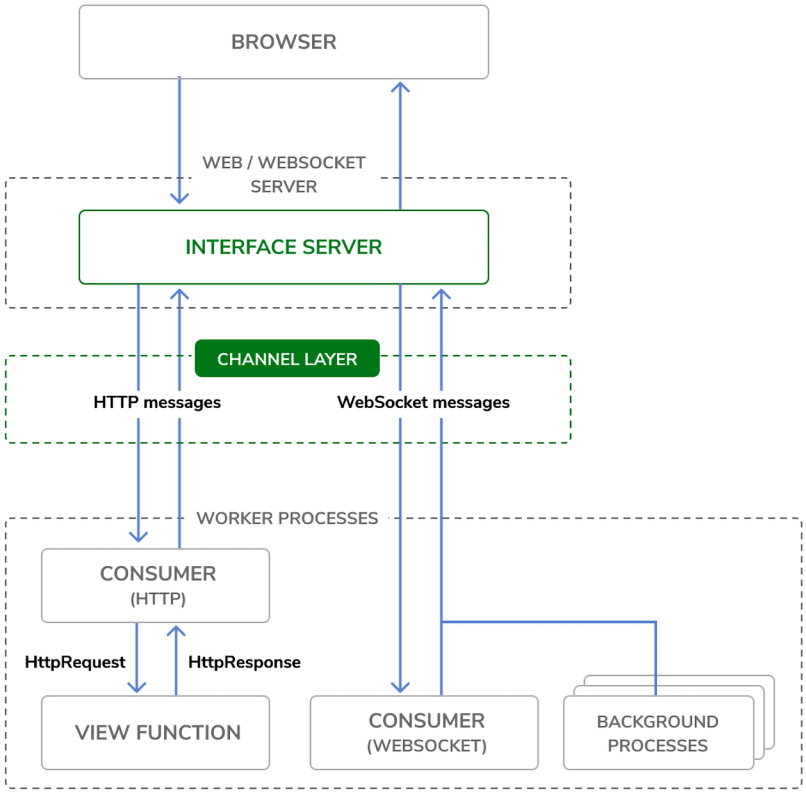


Figure 16.3: The Django Channels request/response cycle

Channels replaces Django's request/response cycle with messages that are sent across channels. HTTP requests are still routed to view functions using Django, but they get routed over channels. This allows for WebSockets message handling as well, where you have producers and consumers that exchange messages across a channel layer. Channels preserves Django's synchronous architecture, allowing you to choose between writing synchronous code and asynchronous code, or a combination of both.

Installing Channels

You are going to add Channels to your project and set up the required basic ASGI application routing for it to manage HTTP requests.

Install Channels in your virtual environment with the following command:

```
pip install channels==3.0.5
```

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

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

The `channels` application is now activated in your project.

Channels expects you to define a single root application that will be executed for all requests. You can define the root application by adding the `ASGI_APPLICATION` setting to your project. This is similar to the `ROOT_URLCONF` setting that points to the base URL patterns of your project. You can place the root application anywhere in your project, but it is recommended to put it in a project-level file. You can add your root routing configuration to the `asgi.py` file directly, where the ASGI application will be defined.

Edit the `asgi.py` file in the `educa` project directory and add the following code highlighted in bold:

```
import os

from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
    'http': django_asgi_app,
})
```

In the previous code, you define the main ASGI application that will be executed when serving the Django project through ASGI. You use the `ProtocolTypeRouter` class provided by Channels as the main entry point of your routing system. `ProtocolTypeRouter` takes a dictionary that maps communication types like `http` or `websocket` to ASGI applications. You instantiate this class with the default application for the HTTP protocol. Later, you will add a protocol for the WebSocket.

Add the following line to the `settings.py` file of your project:

```
ASGI_APPLICATION = 'educa.routing.application'
```

The `ASGI_APPLICATION` setting is used by Channels to locate the root routing configuration.

When Channels is added to the `INSTALLED_APPS` setting, it takes control over the `runserver` command, replacing the standard Django development server. Besides handling URL routing to Django views for synchronous requests, the Channels development server also manages routes to WebSocket consumers.

Start the development server using the following command:

```
python manage.py runserver
```

You will see output similar to the following:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
May 30, 2022 - 08:02:57
Django version 4.0.4, using settings 'educa.settings'
Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Check that the output contains the line `Starting ASGI/Channels version 3.0.4 development server`. This line confirms that you are using the Channels development server, which is capable of managing synchronous and asynchronous requests, instead of the standard Django development server. HTTP requests continue to behave the same as before, but they get routed over Channels.

Now that Channels is installed in your project, you can build the chat server for courses. To implement the chat server for your project, you will need to take the following steps:

1. **Set up a consumer**: Consumers are individual pieces of code that can handle WebSockets in a very similar way to traditional HTTP views. You will build a consumer to read and write messages to a communication channel.
2. **Configure routing**: Channels provides routing classes that allow you to combine and stack your consumers. You will configure URL routing for your chat consumer.
3. **Implement a WebSocket client**: When the student accesses the chat room, you will connect to the WebSocket from the browser and send or receive messages using JavaScript.
4. **Enable a channel layer**: Channel layers allow you to talk between different instances of an application. They're a useful part of making a distributed real-time application. You will set up a channel layer using Redis.

Let's start by writing your own consumer to handle connecting to a WebSocket, receiving and sending messages, and disconnecting.

Writing a consumer

Consumers are the equivalent of Django views for asynchronous applications. As mentioned, they handle WebSockets in a very similar way to how traditional views handle HTTP requests. Consumers are ASGI applications that can handle messages, notifications, and other things. Unlike Django views, consumers are built for long-running communication. URLs are mapped to consumers through routing classes that allow you to combine and stack consumers.

Let's implement a basic consumer that can accept WebSocket connections and echoes every message it receives from the WebSocket back to it. This initial functionality will allow the student to send messages to the consumer and receive back the messages it sends.

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

```
import json
from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):
    def connect(self):
        # accept connection
        self.accept()

    def disconnect(self, close_code):
        pass

    # receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']
        # send message to WebSocket
        self.send(text_data=json.dumps({'message': message}))
```

This is the `ChatConsumer` consumer. This class inherits from the Channels `WebsocketConsumer` class to implement a basic WebSocket consumer. In this consumer, you implement the following methods:

- `connnect()`: Called when a new connection is received. You accept any connection with `self.accept()`. You can also reject a connection by calling `self.close()`.
- `disconnect()`: Called when the socket closes. You use `pass` because you don't need to implement any action when a client closes the connection.
- `receive()`: Called whenever data is received. You expect text to be received as `text_data` (this could also be `binary_data` for binary data). You treat the text data received as JSON. Therefore, you use `json.loads()` to load the received JSON data into a Python dictionary. You access the `message` key, which you expect to be present in the JSON structure received. To echo the message, you send the message back to the WebSocket with `self.send()`, transforming it into JSON format again through `json.dumps()`.

The initial version of your `ChatConsumer` consumer accepts any WebSocket connection and echoes to the WebSocket client every message it receives. Note that the consumer does not broadcast messages to other clients yet. You will build this functionality by implementing a channel layer later.

Routing

You need to define a URL to route connections to the `ChatConsumer` consumer you have implemented. Channels provides routing classes that allow you to combine and stack consumers to dispatch based on what the connection is. You can think of them as the URL routing system of Django for asynchronous applications.

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

```
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
    re_path(r'ws/chat/room/(?P<course_id>\d+)/$',
            consumers.ChatConsumer.as_asgi()),
]
```

In this code, you map a URL pattern with the `ChatConsumer` class that you defined in the `chat/consumers.py` file. You use Django's `re_path` to define the path with regular expressions. You use the `re_path` function instead of the common `path` function because of the limitations of Channels' URL routing. The URL includes an integer parameter called `course_id`. This parameter will be available in the scope of the consumer and will allow you to identify the course chat room that the user is connecting to. You call the `as_asgi()` method of the `consumer` class in order to get an ASGI application that will instantiate an instance of the consumer for each user connection. This behavior is similar to Django's `as_view()` method for class-based views.

It is a good practice to prepend WebSocket URLs with `/ws/` to differentiate them from URLs used for standard synchronous HTTP requests. This also simplifies the production setup when an HTTP server routes requests based on the path.

Edit the global `asgi.py` file located next to the `settings.py` file so that it looks like this:

```
import os

from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
import chat.routing
```

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
    'http': django_asgi_app,
    'websocket': AuthMiddlewareStack(
        URLRouter(chat.routing.websocket_urlpatterns)
    ),
})
```

In this code, you add a new route for the `websocket` protocol. You use `URLRouter` to map `websocket` connections to the URL patterns defined in the `websocket_urlpatterns` list of the `chat` application `routing.py` file. You also use `AuthMiddlewareStack`. The `AuthMiddlewareStack` class provided by Channels supports standard Django authentication, where the user details are stored in the session. Later, you will access the user instance in the scope of the consumer to identify the user who sends a message.

Implementing the WebSocket client

So far, you have created the `course_chat_room` view and its corresponding template for students to access the course chat room. You have implemented a WebSocket consumer for the chat server and tied it with URL routing. Now, you need to build a WebSocket client to establish a connection with the WebSocket in the course chat room template and be able to send/receive messages.

You are going to implement the WebSocket client with JavaScript to open and maintain a connection in the browser. You will interact with the **Document Object Model** (**DOM**) using JavaScript.

Edit the `chat/room.html` template of the `chat` application and modify the `include_js` and `domready` blocks, as follows:

```
{% block include_js %}
  {{ course.id|json_script:"course-id" }}
{% endblock %}

{% block domready %}
  const courseId = JSON.parse(
    document.getElementById('course-id').textContent
  );
  const url = 'ws://' + window.location.host +
              '/ws/chat/room/' + courseId + '/';
  const chatSocket = new WebSocket(url);
{% endblock %}
```

In the `include_js` block, you use the `json_script` template filter to securely use the value of `course.id` with JavaScript. The `json_script` template filter provided by Django outputs a Python object as JSON, wrapped in a `<script>` tag, so that you can safely use it with JavaScript. The code `{{ course.id|json_script:"course-id" }}` is rendered as `<script id="course-id" type="application/json">6</script>`. This value is then retrieved in the `domready` block by parsing the content of the element with `id="course-id"` using `JSON.parse()`. This is the safe way to use Python objects in JavaScript.

You can find more information about the `json_script` template filter at `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#json-script`.

In the `domready` block, you define an URL with the WebSocket protocol, which looks like `ws://` (or `wss://` for secure WebSockets, just like `https://`). You build the URL using the current location of the browser, which you obtain from `window.location.host`. The rest of the URL is built with the path for the chat room URL pattern that you defined in the `routing.py` file of the `chat` application.

You write the URL instead of building it with a resolver because Channels does not provide a way to reverse URLs. You use the current course `ID` to generate the URL for the current course and store the URL in a new constant named `url`.

You then open a WebSocket connection to the stored URL using `new WebSocket(url)`. You assign the instantiated WebSocket client object to the new constant `chatSocket`.

You have created a WebSocket consumer, you have included routing for it, and you have implemented a basic WebSocket client. Let's try the initial version of your chat.

Start the development server using the following command:

```
python manage.py runserver
```

Open the URL `http://127.0.0.1:8000/chat/room/1/` in your browser, replacing `1` with the `id` of an existing course in the database. Take a look at the console output. Besides the HTTP `GET` requests for the page and its static files, you should see two lines including `WebSocket HANDSHAKING` and `WebSocket CONNECT`, like the following output:

```
HTTP GET /chat/room/1/ 200 [0.02, 127.0.0.1:57141]
HTTP GET /static/css/base.css 200 [0.01, 127.0.0.1:57141]
WebSocket HANDSHAKING /ws/chat/room/1/ [127.0.0.1:57144]
WebSocket CONNECT /ws/chat/room/1/ [127.0.0.1:57144]
```

The Channels development server listens for incoming socket connections using a standard TCP socket. The handshake is the bridge from HTTP to WebSockets. In the handshake, details of the connection are negotiated and either party can close the connection before completion. Remember that you are using `self.accept()` to accept any connection in the `connect()` method of the `ChatConsumer` class, implemented in the `consumers.py` file of the `chat` application. The connection is accepted, and therefore, you see the `WebSocket CONNECT` message in the console.

If you use the browser developer tools to track network connections, you can also see information for the WebSocket connection that has been established.

It should look like *Figure 16.4*:

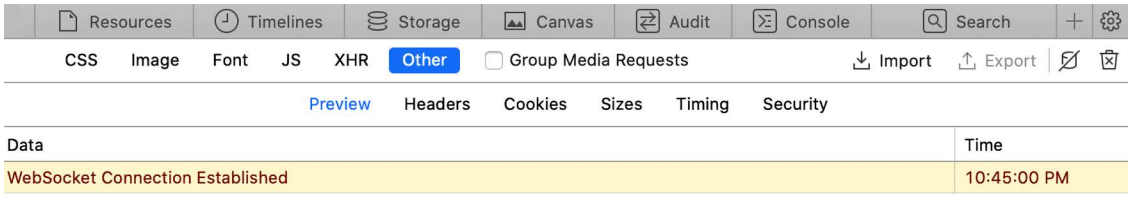


Figure 16.4: The browser developer tools showing that the WebSocket connection has been established

Now that you can connect to the WebSocket, it's time to interact with it. You will implement the methods to handle common events, such as receiving a message and closing the connection. Edit the `chat/room.html` template of the `chat` application and modify the `domready` block, as follows:

```
{% block domready %}
  const courseId = JSON.parse(
    document.getElementById('course-id').textContent
  );
  const url = 'ws://' + window.location.host +
              '/ws/chat/room/' + courseId + '/';
  const chatSocket = new WebSocket(url);

  chatSocket.onmessage = function(event) {
    const data = JSON.parse(event.data);
    const chat = document.getElementById('chat');

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

  chatSocket.onclose = function(event) {
    console.error('Chat socket closed unexpectedly');
  };

{% endblock %}
```

In this code, you define the following events for the WebSocket client:

- `onmessage`: Fired when data is received through the WebSocket. You parse the message, which you expect in JSON format, and access its `message` attribute. You then append a new `<div>` element with the message received to the HTML element with the `chat` ID. This will add new messages to the chat log, while keeping all previous messages that have been added to the log. You scroll the chat log `<div>` to the bottom to ensure that the new message gets visibility. You achieve this by scrolling to the total scrollable height of the chat log, which can be obtained by accessing its `scrollHeight` attribute.

- `onclose`: Fired when the connection with the WebSocket is closed. You don't expect to close the connection, and therefore, you write the error `Chat socket closed unexpectedly` to the console log if this happens.

You have implemented the action to display the message when a new message is received. You need to implement the functionality to send messages to the socket as well.

Edit the `chat/room.html` template of the `chat` application and add the following JavaScript code to the bottom of the `domready` block:

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

submitButton.addEventListener('click', function(event) {
  const message = input.value;
  if(message) {
    // send message in JSON format
    chatSocket.send(JSON.stringify({'message': message}));
    // clear input
    input.innerHTML = '';
    input.focus();
  }
});
```

In this code, you define an event listener for the `click` event of the submit button, which you select by its ID `chat-message-submit`. When the button is clicked, you perform the following actions:

1. You read the message entered by the user from the value of the text input element with the ID `chat-message-input`.
2. You check whether the message has any content with `if(message)` .
3. If the user has entered a message, you form JSON content such as `{'message': 'string entered by the user'}` by using `JSON.stringify()`.
4. You send the JSON content through the WebSocket, calling the `send()` method of `chatSocket` client.
5. You clear the contents of the text input by setting its value to an empty string with `input.innerHTML = ''`.
6. You return the focus to the text input with `input.focus()` so that the user can write a new message straightaway.

The user is now able to send messages using the text input and by clicking the submit button.

To improve the user experience, you will give focus to the text input as soon as the page loads so that the user can type directly in it. You will also capture keyboard keypress events to identify the *Enter* key and fire the `click` event on the submit button. The user will be able to either click the button or press the *Enter* key to send a message.

Edit the `chat/room.html` template of the `chat` application and add the following JavaScript code to the bottom of the `domready` block:

```
input.addEventListener('keypress', function(event) {
    if (event.key === 'Enter') {
      // cancel the default action, if needed
      event.preventDefault();
      // trigger click event on button
      submitButton.click();
    }
  });

input.focus();
```

In this code, you also define a function for the `keypress` event of the `input` element. For any key that the user presses, you check whether its key is `Enter`. You prevent the default behavior for this key with `event.preventDefault()`. If the *Enter* key is pressed, you fire the `click` event on the submit button to send the message to the WebSocket.

Outside of the event handler, in the main JavaScript code for the `domready` block, you give the focus to the text input with `input.focus()`. By doing so, when the DOM is loaded, the focus will be set on the `input` element for the user to type a message.

The `domready` block of the `chat/room.html` template should now look as follows:

```
{% block domready %}
  const courseId = JSON.parse(
    document.getElementById('course-id').textContent
  );
  const url = 'ws://' + window.location.host +
              '/ws/chat/room/' + courseId + '/';
  const chatSocket = new WebSocket(url);

  chatSocket.onmessage = function(event) {
    const data = JSON.parse(event.data);
    const chat = document.getElementById('chat');

    chat.innerHTML += '<div class="message">' +
                      data.message + '</div>';
    chat.scrollTop = chat.scrollHeight;
  };
```

```
  chatSocket.onclose = function(event) {
    console.error('Chat socket closed unexpectedly');
  };

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

  submitButton.addEventListener('click', function(event) {
    const message = input.value;
    if(message) {
      // send message in JSON format
      chatSocket.send(JSON.stringify({'message': message}));
      // clear input
      input.value = '';
      input.focus();
    }
  });

  input.addEventListener('keypress', function(event) {
    if (event.key === 'Enter') {
      // cancel the default action, if needed
      event.preventDefault();
      // trigger click event on button
      submitButton.click();
    }
  });

  input.focus();
{% endblock %}
```

Open the URL `http://127.0.0.1:8000/chat/room/1/` in your browser, replacing `1` with the `id` of an existing course in the database. With a logged-in user who is enrolled on the course, write some text in the input field and click the **SEND** button or press the *Enter* key.

You will see that your message appears in the chat log:

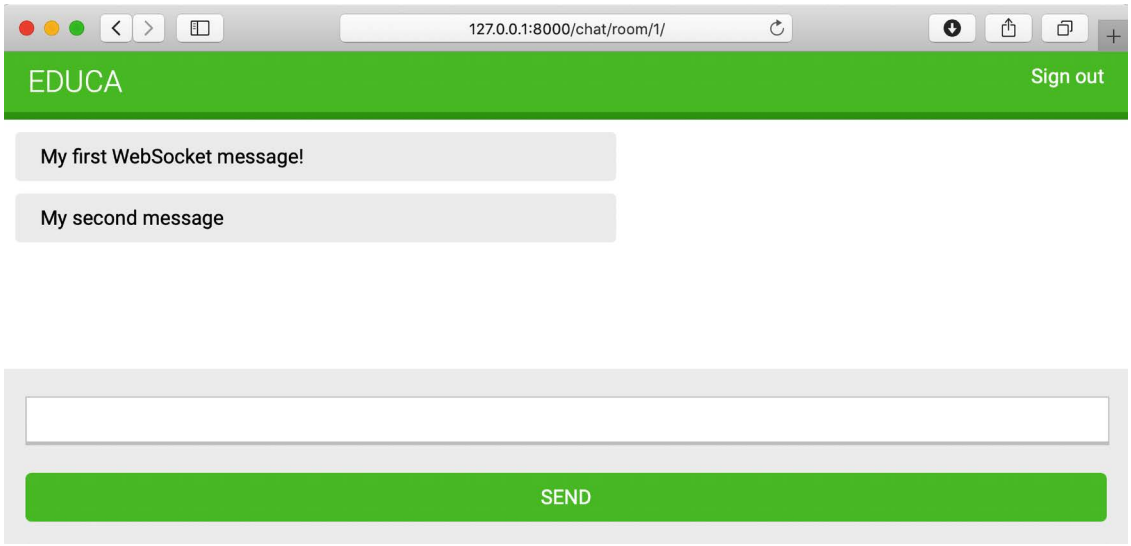


Figure 16.5: The chat room page, including messages sent through the WebSocket

Great! The message has been sent through the WebSocket and the `ChatConsumer` consumer has received the message and has sent it back through the WebSocket. The `chatSocket` client has received a message event and the `onmessage` function has been fired, adding the message to the chat log.

You have implemented the functionality with a WebSocket consumer and a WebSocket client to establish client/server communication and can send or receive events. However, the chat server is not able to broadcast messages to other clients. If you open a second browser tab and enter a message, the message will not appear on the first tab. In order to build communication between consumers, you have to enable a channel layer.

Enabling a channel layer

Channel layers allow you to communicate between different instances of an application. A channel layer is the transport mechanism that allows multiple consumer instances to communicate with each other and with other parts of Django.

In your chat server, you plan to have multiple instances of the `ChatConsumer` consumer for the same course chat room. Each student who joins the chat room will instantiate the WebSocket client in their browser, and that will open a connection with an instance of the WebSocket consumer. You need a common channel layer to distribute messages between consumers.

Channels and groups

Channel layers provide two abstractions to manage communications: channels and groups:

- **Channel**: You can think of a channel as an inbox where messages can be sent to or as a task queue. Each channel has a name. Messages are sent to a channel by anyone who knows the channel name and then given to consumers listening on that channel.
- **Group**: Multiple channels can be grouped into a group. Each group has a name. A channel can be added or removed from a group by anyone who knows the group name. Using the group name, you can also send a message to all channels in the group.

You will work with channel groups to implement the chat server. By creating a channel group for each course chat room, the `ChatConsumer` instances will be able to communicate with each other.

Setting up a channel layer with Redis

Redis is the preferred option for a channel layer, though Channels has support for other types of channel layers. Redis works as the communication store for the channel layer. Remember that you already used Redis in *Chapter 7*, *Tracking User Actions*, *Chapter 10*, *Extending Your Shop*, and *Chapter 14*, *Rendering and Caching Content*.

If you haven't installed Redis yet, you can find installation instructions in *Chapter 7*, *Tracking User Actions*.

To use Redis as a channel layer, you have to install the `channels-redis` package. Install `channels-redis` in your virtual environment with the following command:

```
pip install channels-redis==3.4.1
```

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

```
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('127.0.0.1', 6379)],
        },
    },
}
```

The `CHANNEL_LAYERS` setting defines the configuration for the channel layers available to the project. You define a default channel layer using the `RedisChannelLayer` backend provided by `channels-redis` and specify the host `127.0.0.1` and the port `6379`, on which Redis is running.

Let's try the channel layer. Initialize the Redis Docker container using the following command:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

If you want to run the command in the background (in detached mode) you can use the `-d` option.

Open the Django shell using the following command from the project directory:

```
python manage.py shell
```

To verify that the channel layer can communicate with Redis, write the following code to send a message to a test channel named `test_channel` and receive it back:

```
>>> import channels.layers
>>> from asgiref.sync import async_to_sync
>>> channel_layer = channels.layers.get_channel_layer()
>>> async_to_sync(channel_layer.send)('test_channel', {'message': 'hello'})
>>> async_to_sync(channel_layer.receive)('test_channel')
```

You should get the following output:

```
{'message': 'hello'}
```

In the previous code, you send a message to a test channel through the channel layer, and then you retrieve it from the channel layer. The channel layer is communicating successfully with Redis.

Updating the consumer to broadcast messages

Let's edit the `ChatConsumer` consumer to use the channel layer. You will use a channel group for each course chat room. Therefore, you will use the course `id` to build the group name. `ChatConsumer` instances will know the group name and will be able to communicate with each other.

Edit the `consumers.py` file of the `chat` application, import the `async_to_sync()` function, and modify the `connect()` method of the `ChatConsumer` class, as follows:

```
import json
from channels.generic.websocket import WebsocketConsumer
from asgiref.sync import async_to_sync

class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.id = self.scope['url_route']['kwargs']['course_id']
        self.room_group_name = f'chat_{self.id}'
        # join room group
        async_to_sync(self.channel_layer.group_add)(
            self.room_group_name,
            self.channel_name
        )
        # accept connection
        self.accept()
    # ...
```

In this code, you import the `async_to_sync()` helper function to wrap calls to asynchronous channel layer methods. `ChatConsumer` is a synchronous `WebsocketConsumer` consumer, but it needs to call asynchronous methods of the channel layer.

In the new `connect()` method, you perform the following tasks:

1. You retrieve the course `id` from the scope to know the course that the chat room is associated with. You access `self.scope['url_route']['kwargs']['course_id']` to retrieve the `course_id` parameter from the URL. Every consumer has a scope with information about its connection, arguments passed by the URL, and the authenticated user, if any.
2. You build the group name with the `id` of the course that the group corresponds to. Remember that you will have a channel group for each course chat room. You store the group name in the `room_group_name` attribute of the consumer.
3. You join the group by adding the current channel to the group. You obtain the channel name from the `channel_name` attribute of the consumer. You use the `group_add` method of the channel layer to add the channel to the group. You use the `async_to_sync()` wrapper to use the channel layer asynchronous method.
4. You keep the `self.accept()` call to accept the WebSocket connection.

When the `ChatConsumer` consumer receives a new WebSocket connection, it adds the channel to the group associated with the course in its scope. The consumer is now able to receive any messages sent to the group.

In the same `consumers.py` file, modify the `disconnect()` method of the `ChatConsumer` class, as follows:

```
class ChatConsumer(WebsocketConsumer):
    # ...
    def disconnect(self, close_code):
        # leave room group
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name,
            self.channel_name
        )
    # ...
```

When the connection is closed, you call the `group_discard()` method of the channel layer to leave the group. You use the `async_to_sync()` wrapper to use the channel layer asynchronous method.

In the same `consumers.py` file, modify the `receive()` method of the `ChatConsumer` class, as follows:

```
class ChatConsumer(WebsocketConsumer):
    # ...
    # receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
```

```
        message = text_data_json['message']
        # send message to room group
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
            }
        )
```

When you receive a message from the WebSocket connection, instead of sending the message to the associated channel, you send the message to the group. You do this by calling the `group_send()` method of the channel layer. You use the `async_to_sync()` wrapper to use the channel layer asynchronous method. You pass the following information in the event sent to the group:

- `type`: The event type. This is a special key that corresponds to the name of the method that should be invoked on consumers that receive the event. You can implement a method in the consumer named the same as the message type so that it gets executed every time a message with that specific type is received.
- `message`: The actual message you are sending.

In the same `consumers.py` file, add a new `chat_message()` method in the `ChatConsumer` class, as follows:

```
class ChatConsumer(WebsocketConsumer):
    # ...
    # receive message from room group
    def chat_message(self, event):
        # send message to WebSocket
        self.send(text_data=json.dumps(event))
```

You name this method `chat_message()` to match the `type` key that is sent to the channel group when a message is received from the WebSocket. When a message with type `chat_message` is sent to the group, all consumers subscribed to the group will receive the message and will execute the `chat_message()` method. In the `chat_message()` method, you send the event message received to the WebSocket.

The complete `consumers.py` file should now look like this:

```
import json
from channels.generic.websocket import WebsocketConsumer
from asgiref.sync import async_to_sync

class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.id = self.scope['url_route']['kwargs']['course_id']
        self.room_group_name = f'chat_{self.id}'
```

```
        # join room group
        async_to_sync(self.channel_layer.group_add)(
            self.room_group_name,
            self.channel_name
        )
        # accept connection
        self.accept()

    def disconnect(self, close_code):
        # leave room group
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name,
            self.channel_name
        )

    # receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']
        # send message to room group
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
            }
        )

    # receive message from room group
    def chat_message(self, event):
        # send message to WebSocket
        self.send(text_data=json.dumps(event))
```

You have implemented a channel layer in `ChatConsumer`, allowing consumers to broadcast messages and communicate with each other.

Run the development server with the following command:

```
python manage.py runserver
```

Open the URL `http://127.0.0.1:8000/chat/room/1/` in your browser, replacing `1` with the `id` of an existing course in the database. Write a message and send it. Then, open a second browser window and access the same URL. Send a message from each browser window.

The result should look like this:

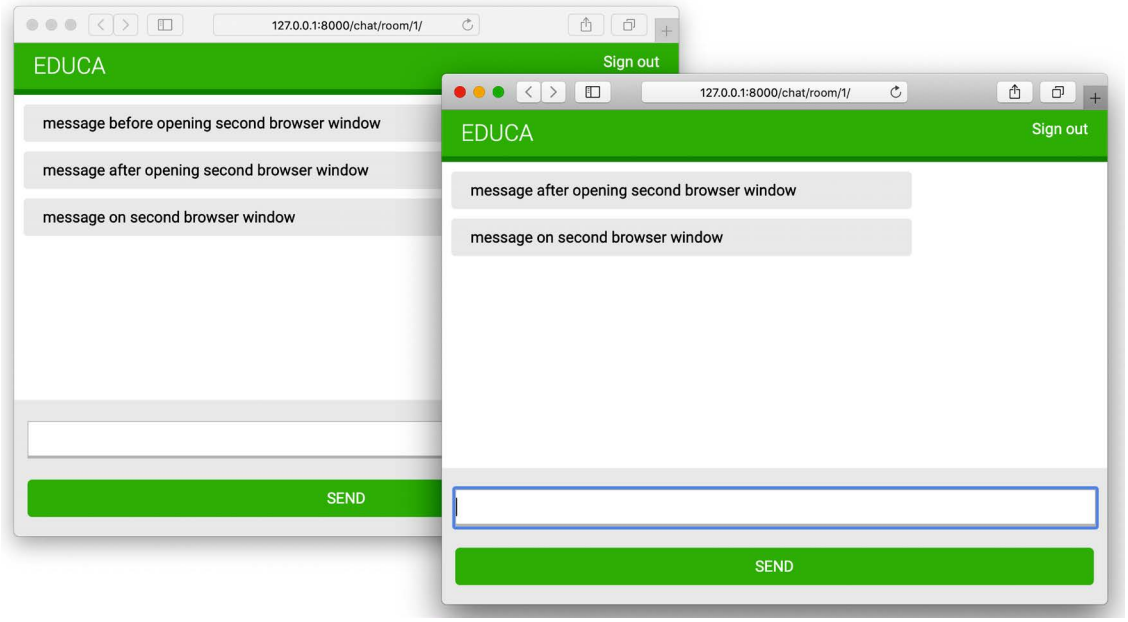


Figure 16.6: The chat room page with messages sent from different browser windows

You will see that the first message is only displayed in the first browser window. When you open a second browser window, messages sent in any of the browser windows are displayed in both of them. When you open a new browser window and access the chat room URL, a new WebSocket connection is established between the JavaScript WebSocket client in the browser and the WebSocket consumer in the server. Each channel gets added to the group associated with the course ID and passed through the URL to the consumer. Messages are sent to the group and received by all consumers.

Adding context to the messages

Now that messages can be exchanged between all users in a chat room, you probably want to display who sent which message and when it was sent. Let's add some context to the messages.

Edit the `consumers.py` file of the `chat` application and implement the following changes:

```
import json
from channels.generic.websocket import WebsocketConsumer
from asgiref.sync import async_to_sync
from django.utils import timezone

class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.user = self.scope['user']
```

```
        self.id = self.scope['url_route']['kwargs']['course_id']
        self.room_group_name = f'chat_{self.id}'
        # join room group
        async_to_sync(self.channel_layer.group_add)(
            self.room_group_name,
            self.channel_name
        )
        # accept connection
        self.accept()

    def disconnect(self, close_code):
        # leave room group
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name,
            self.channel_name
        )

    # receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']
        now = timezone.now()
        # send message to room group
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
                'user': self.user.username,
                'datetime': now.isoformat(),
            }
        )

    # receive message from room group
    def chat_message(self, event):
        # send message to WebSocket
        self.send(text_data=json.dumps(event))
```

You now import the `timezone` module provided by Django. In the `connect()` method of the consumer, you retrieve the current user from the scope with `self.scope['user']` and store them in a new `user` attribute of the consumer. When the consumer receives a message through the WebSocket, it gets the current time using `timezone.now()` and passes the current `user` and `datetime` in ISO 8601 format along with the message in the event sent to the channel group.

Edit the `chat/room.html` template of the `chat` application and add the following line highlighted in bold to the `include_js` block:

```
{% block include_js %}
  {{ course.id|json_script:"course-id" }}
  {{ request.user.username|json_script:"request-user" }}
{% endblock %}
```

Using the `json_script` template, you safely print the username of the request user to use it with JavaScript.

In the `domready` block of the `chat/room.html` template, add the following lines highlighted in bold:

```
{% block domready %}
  const courseId = JSON.parse(
    document.getElementById('course-id').textContent
  );
  const requestUser = JSON.parse(
    document.getElementById('request-user').textContent
  );
  # ...
{% endblock %}
```

In the new code, you safely parse the data of the element with the ID `request-user` and store it in the `requestUser` constant.

Then, in the `domready` block, find the following lines:

```
const data = JSON.parse(e.data);
const chat = document.getElementById('chat');

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

Replace those lines with the following code:

```
const data = JSON.parse(e.data);
const chat = document.getElementById('chat');

const dateOptions = {hour: 'numeric', minute: 'numeric', hour12: true};
const datetime = new Date(data.datetime).toLocaleString('en', dateOptions);
const isMe = data.user === requestUser;
const source = isMe ? 'me' : 'other';
const name = isMe ? 'Me' : data.user;

chat.innerHTML += '<div class="message ' + source + '">' +
                  '<strong>' + name + '</strong> ' +
                  '<span class="date">' + datetime + '</span><br>' +
                  data.message + '</div>';
chat.scrollTop = chat.scrollHeight;
```

In this code, you implement the following changes:

1. You convert the `datetime` received in the message to a JavaScript `Date` object and format it with a specific locale.
2. You compare the username received in the message with two different constants as helpers to identify the user.
3. The constant `source` gets the value `me` if the user sending the message is the current user, or `other` otherwise.
4. The constant `name` gets the value `Me` if the user sending the message is the current user or the name of the user sending the message otherwise. You use it to display the name of the user sending the message.
5. You use the `source` value as a `class` of the main `<div>` message element to differentiate messages sent by the current user from messages sent by others. Different CSS styles are applied based on the `class` attribute. These CSS styles are declared in the `css/base.css` static file.
6. You use the username and the `datetime` in the message that you append to the chat log.

Open the URL `http://127.0.0.1:8000/chat/room/1/` in your browser, replacing `1` with the `id` of an existing course in the database. With a logged-in user who is enrolled on the course, write a message and send it.

Then, open a second browser window in incognito mode to prevent the use of the same session. Log in with a different user, also enrolled on the same course, and send a message.

You will be able to exchange messages using the two different users and see the user and time, with a clear distinction between messages sent by the user and messages sent by others. The conversation between two users should look similar to the following one:

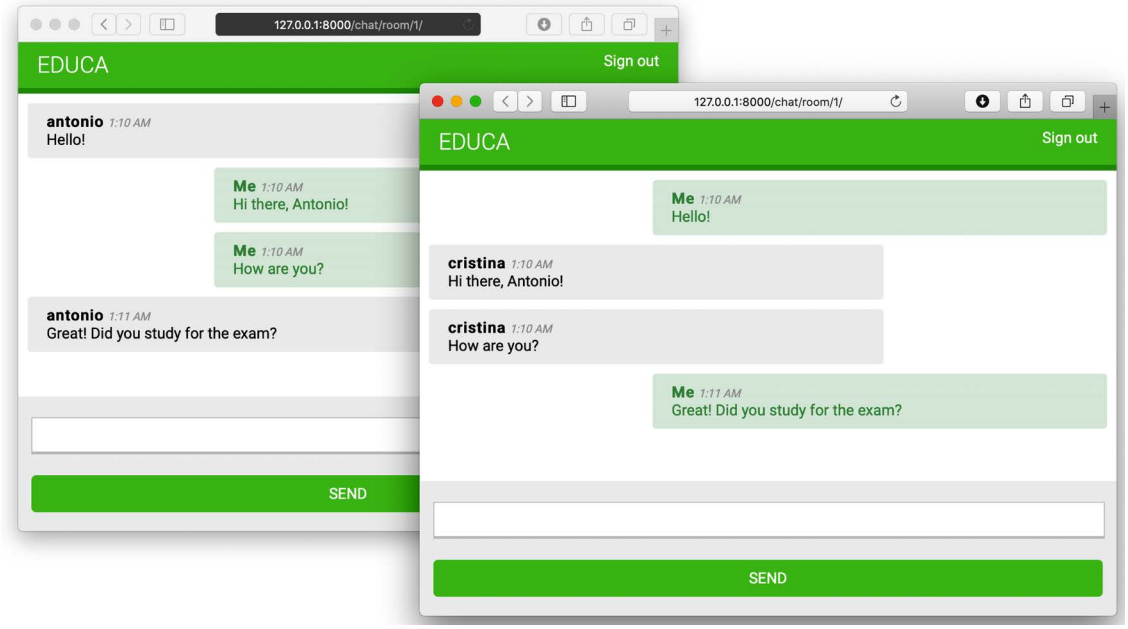


Figure 16.7: The chat room page with messages from two different user sessions

Great! You have built a functional real-time chat application using Channels. Next, you will learn how to improve the chat consumer by making it fully asynchronous.

Modifying the consumer to be fully asynchronous

The `ChatConsumer` you have implemented inherits from the base `WebsocketConsumer` class, which is synchronous. Synchronous consumers are convenient for accessing Django models and calling regular synchronous I/O functions. However, asynchronous consumers perform better, since they don't require additional threads when handling requests. Since you are using the asynchronous channel layer functions, you can easily rewrite the `ChatConsumer` class to be asynchronous.

Edit the `consumers.py` file of the `chat` application and implement the following changes:

```
import json
from channels.generic.websocket import AsyncWebsocketConsumer
from asgiref.sync import async_to_sync
from django.utils import timezone

class ChatConsumer(AsyncWebsocketConsumer):
```

```
    async def connect(self):
        self.user = self.scope['user']
        self.id = self.scope['url_route']['kwargs']['course_id']
        self.room_group_name = 'chat_%s' % self.id
        # join room group
        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )
        # accept connection
        await self.accept()

    async def disconnect(self, close_code):
        # leave room group
        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

    # receive message from WebSocket
    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json['message']
        now = timezone.now()
        # send message to room group
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
                'user': self.user.username,
                'datetime': now.isoformat(),
            }
        )

    # receive message from room group
    async def chat_message(self, event):
        # send message to WebSocket
        await self.send(text_data=json.dumps(event))
```

You have implemented the following changes:

1. The `ChatConsumer` consumer now inherits from the `AsyncWebsocketConsumer` class to implement asynchronous calls
2. You have changed the definition of all methods from `def` to `async def`
3. You use `await` to call asynchronous functions that perform I/O operations
4. You no longer use the `async_to_sync()` helper function when calling methods on the channel layer

Open the URL `http://127.0.0.1:8000/chat/room/1/` with two different browser windows again and verify that the chat server still works. The chat server is now fully asynchronous!

Integrating the chat application with existing views

The chat server is now fully implemented, and students enrolled on a course can communicate with each other. Let's add a link for students to join the chat room for each course.

Edit the `students/course/detail.html` template of the `students` application and add the following `<h3>` HTML element code at the bottom of the `<div class="contents">` element:

```
<div class="contents">
  ...
  <h3>
    <a href="{% url "chat:course_chat_room" object.id %}">
      Course chat room
    </a>
  </h3>
</div>
```

Open the browser and access any course that the student is enrolled on to view the course contents. The sidebar will now contain a **Course chat room** link that points to the course chat room view. If you click on it, you will enter the chat room:

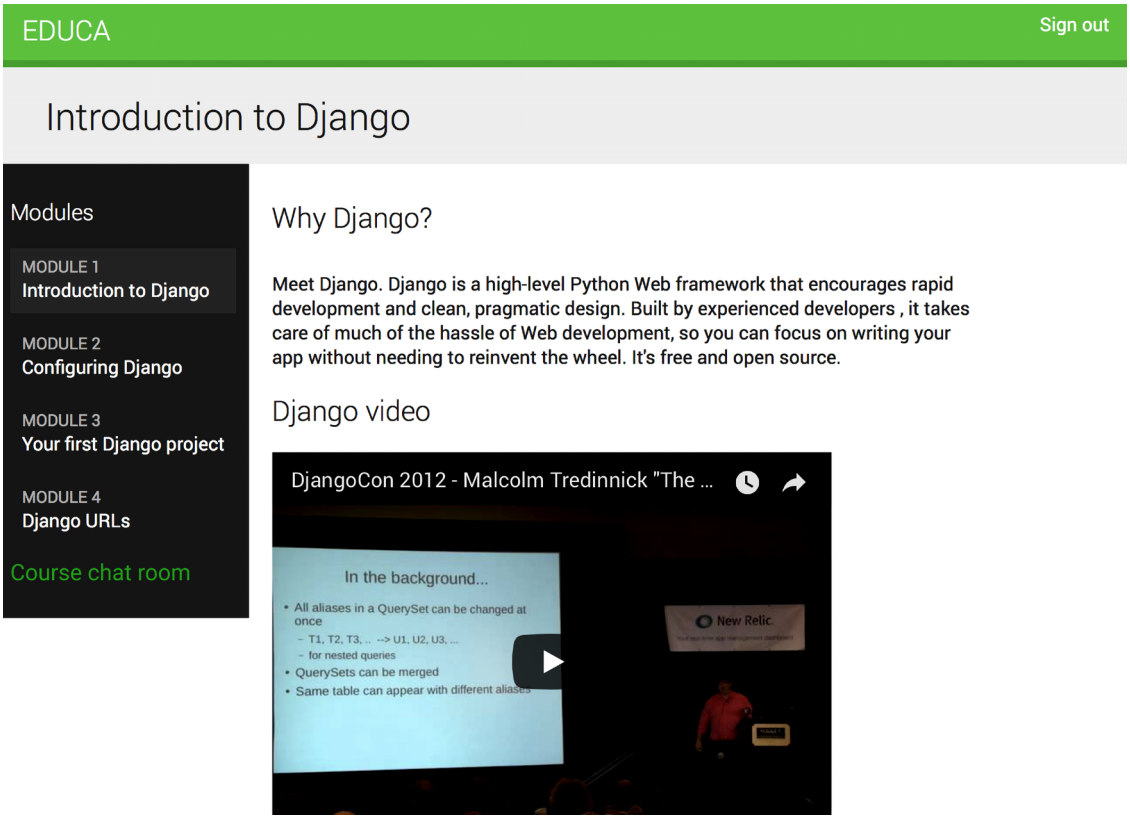


Figure 16.8: The course detail page, including a link to the course chat room

Congratulations! You successfully built your first asynchronous application using Django Channels.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter16`
- Introduction to ASGI – `https://asgi.readthedocs.io/en/latest/introduction.html`
- Django support for asynchronous views – `https://docs.djangoproject.com/en/4.1/topics/async/`
- Django support for asynchronous class-based views – `https://docs.djangoproject.com/en/4.1/topics/class-based-views/#async-class-based-views`
- Django Channels documentation – `https://channels.readthedocs.io/`
- Deploying Django with ASGI – `https://docs.djangoproject.com/en/4.1/howto/deployment/asgi/`
- `json_script` template filter usage – `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#json-script`

Summary

In this chapter, you learned how to create a chat server using Channels. You implemented a WebSocket consumer and client. You also enabled communication between consumers using a channel layer with Redis and modified the consumer to be fully asynchronous.

The next chapter will teach you how to build a production environment for your Django project using NGINX, uWSGI, and Daphne with Docker Compose. You will also learn how to implement custom middleware and create custom management commands.

17

Going Live

In the previous chapter, you built a real-time chat server for students using Django Channels. Now that you have created a fully functional e-learning platform, you need to set up a production environment so that it can be accessed over the internet. Until now, you have been working in a development environment, using the Django development server to run your site. In this chapter, you will learn how to set up a production environment that is able to serve your Django project in a secure and efficient manner.

This chapter will cover the following topics:

- Configuring Django settings for multiple environments
- Using Docker Compose to run multiple services
- Setting up a web server with uWSGI and Django
- Serving PostgreSQL and Redis with Docker Compose
- Using the Django system check framework
- Serving NGINX with Docker
- Serving static assets through NGINX
- Securing connections through TLS/SSL
- Using the Daphne ASGI server for Django Channels
- Creating a custom Django middleware
- Implementing custom Django management commands

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter17`.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `pip install -r requirements.txt`.

Creating a production environment

It's time to deploy your Django project in a production environment. You will start by configuring Django settings for multiple environments, and then you will set up a production environment.

Managing settings for multiple environments

In real-world projects, you will have to deal with multiple environments. You will usually have at least a local environment for development and a production environment for serving your application. You could have other environments as well, such as testing or staging environments.

Some project settings will be common to all environments, but others will be specific to each environment. Usually, you will use a base file that defines common settings, and a settings file per environment that overrides any necessary settings and defines additional ones.

We will manage the following environments:

- `local`: The local environment to run the project on your machine.
- `prod`: The environment for deploying your project on a production server.

Create a `settings/` directory next to the `settings.py` file of the `educa` project. Rename the `settings.py` file to `base.py` and move it into the new `settings/` directory.

Create the following additional files inside the `settings/` folder so that the new directory looks as follows:

```
settings/
    __init__.py
    base.py
    local.py
    prod.py
```

These files are as follows:

- `base.py`: The base settings file that contains common settings (previously `settings.py`)
- `local.py`: Custom settings for your local environment
- `prod.py`: Custom settings for the production environment

You have moved the settings files to a directory one level below, so you need to update the `BASE_DIR` setting in the `settings/base.py` file to point to the main project directory.

When handling multiple environments, create a base settings file and a settings file for each environment. Environment settings files should inherit the common settings and override environment-specific settings.

Edit the `settings/base.py` file and replace the following line:

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

with the following one:

```
BASE_DIR = Path(__file__).resolve().parent.parent.parent
```

You point to one directory above by adding `.parent` to the `BASE_DIR` path. Let's configure the settings for the local environment.

Local environment settings

Instead of using a default configuration for the `DEBUG` and `DATABASES` settings, you will define them for each environment explicitly. These settings will be environment specific. Edit the `educa/settings/local.py` file and add the following lines:

```
from .base import *

DEBUG = True

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

This is the settings file for your local environment. In this file, you import all settings defined in the `base.py` file, and you define the `DEBUG` and `DATABASES` settings for this environment. The `DEBUG` and `DATABASES` settings remain the same as you have been using for development.

Now remove the `DATABASES` and `DEBUG` settings from the `base.py` settings file.

Django management commands won't automatically detect the settings file to use because the project settings file is not the default `settings.py` file. When running management commands, you need to indicate the settings module to use by adding a `--settings` option, as follows:

```
python manage.py runserver --settings=educa.settings.local
```

Next, we are going to validate the project and the local environment configuration.

Running the local environment

Let's run the local environment using the new settings structure. Make sure Redis is running or start the Redis Docker container in a shell with the following command:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Run the following management command in another shell, from the project directory:

```
python manage.py runserver --settings=educa.settings.local
```

Open `http://127.0.0.1:8000/` in your browser and check that the site loads correctly. You are now serving your site using the settings for the `local` environment.

If don't want to pass the `--settings` option every time you run a management command, you can define the `DJANGO_SETTINGS_MODULE` environment variable. Django will use it to identify the settings module to use. If you are using Linux or macOS, you can define the environment variable by executing the following command in the shell:

```
export DJANGO_SETTINGS_MODULE=educa.settings.local
```

If you are using Windows, you can execute the following command in the shell:

```
set DJANGO_SETTINGS_MODULE=educa.settings.local
```

Any management command you execute after will use the settings defined in the `DJANGO_SETTINGS_MODULE` environment variable.

Stop the Django development server from the shell by pressing the keys *Ctrl* + *C* and stop the Redis Docker container from the shell by also pressing the keys *Ctrl* + *C*.

The local environment works well. Let's prepare the settings for the production environment.

Production environment settings

Let's start by adding initial settings for the production environment. Edit the `educa/settings/prod.py` file and make it look as follows:

```
from .base import *

DEBUG = False

ADMINS = [
    ('Antonio M', 'email@mydomain.com'),
]

ALLOWED_HOSTS = ['*']

DATABASES = {
    'default': {
    }
}
```

These are the settings for the production environment:

- `DEBUG`: Setting `DEBUG` to `False` is necessary for any production environment. Failing to do so will result in the traceback information and sensitive configuration data being exposed to everyone.

- `ADMINS`: When `DEBUG` is `False` and a view raises an exception, all information will be sent by email to the people listed in the `ADMINS` setting. Make sure that you replace the name/email tuple with your own information.
- `ALLOWED_HOSTS`: For security reasons, Django will only allow the hosts included in this list to serve the project. For now, you allow all hosts by using the asterisk symbol, `*`. You will limit the hosts that can be used for serving the project later.
- `DATABASES`: You keep `default` database settings empty because you will configure the production database later.

Over the next sections of this chapter, you will complete the settings file for your production environment.

You have successfully organized settings for handling multiple environments. Now you will build a complete production environment by setting up different services with Docker.

Using Docker Compose

Docker allows you to build, deploy, and run application containers. A Docker container combines application source code with operating system libraries and dependencies required to run the application. By using application containers, you can improve your application portability. You are already using a Redis Docker image to serve Redis in your local environment. This Docker image contains everything needed to run Redis and allows you to run it seamlessly on your machine. For the production environment, you will use Docker Compose to build and run different Docker containers.

Docker Compose is a tool for defining and running multi-container applications. You can create a configuration file to define the different services and use a single command to start all services from your configuration. You can find information about Docker Compose at `https://docs.docker.com/compose/`.

For the production environment, you will create a distributed application that runs across multiple Docker containers. Each Docker container will run a different service. You will initially define the following three services and you will add additional services in the next sections:

- Web service: A web server to serve the Django project
- Database service: A database service to run PostgreSQL
- Cache service: A service to run Redis

Let's start by installing Docker Compose.

Installing Docker Compose

You can run Docker Compose on macOS, 64-bit Linux, and Windows. The fastest way to install Docker Compose is by installing Docker Desktop. The installation includes Docker Engine, the command-line interface, and the Docker Compose plugin.

Install Docker Desktop by following the instructions at `https://docs.docker.com/compose/install/compose-desktop/`.

Open the Docker Desktop application and click on **Containers**. It will look as follows:

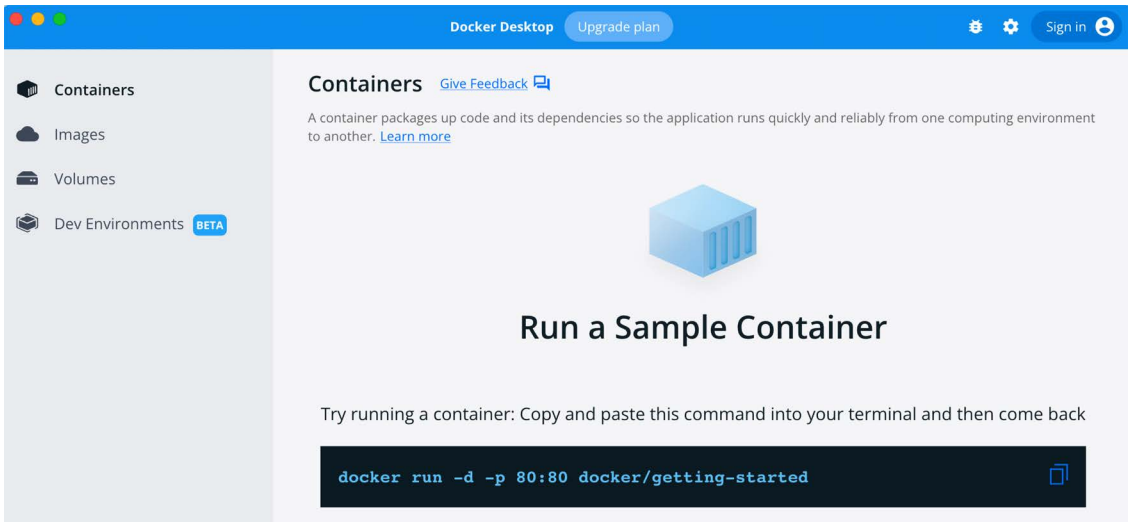


Figure 17.1: The Docker Desktop interface

After installing Docker Compose, you will need to create a Docker image for your Django project.

Creating a Dockerfile

You need to create a Docker image to run the Django project. A `Dockerfile` is a text file that contains the commands for Docker to assemble a Docker image. You will prepare a `Dockerfile` with the commands to build the Docker image for the Django project.

Next to the `educa` project directory, create a new file and name it `Dockerfile`. Add the following code to the new file:

```
# Pull official base Python Docker image
FROM python:3.10.6

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Set work directory
WORKDIR /code

# Install dependencies
RUN pip install --upgrade pip
```

```
COPY requirements.txt /code/
RUN pip install -r requirements.txt

# Copy the Django project
COPY . /code/
```

This code performs the following tasks:

1. The Python 3.10.6 parent Docker image is used. You can find the official Python Docker image at `https://hub.docker.com/_/python`.
2. The following environment variables are set:
    a. `PYTHONDONTWRITEBYTECODE`: Prevents Python from writing out `pyc` files.
    b. `PYTHONUNBUFFERED`: Ensures that the Python `stdout` and `stderr` streams are sent straight to the terminal without first being buffered.
3. The `WORKDIR` command is used to define the working directory of the image.
4. The `pip` package of the image is upgraded.
5. The `requirements.txt` file is copied to the `code` directory of the parent Python image.
6. The Python packages in `requirements.txt` are installed in the image using `pip`.
7. The Django project source code is copied from the local directory to the `code` directory of the image.

With this `Dockerfile`, you have defined how the Docker image to serve Django will be assembled. You can find the `Dockerfile` reference at `https://docs.docker.com/engine/reference/builder/`.

Adding the Python requirements

A `requirements.txt` file is used in the `Dockerfile` you created to install all necessary Python packages for the project.

Next to the `educa` project directory, create a new file and name it `requirements.txt`. You may have already created this file before and copied the content for the `requirements.txt` file from `https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/requirements.txt`. If you haven't done so, add the following lines to the newly created `requirements.txt` file:

```
asgiref==3.5.2
Django~=4.1
Pillow==9.2.0
sqlparse==0.4.2
django-braces==1.15.0
django-embed-video==1.4.4
pymemcache==3.5.2
django-debug-toolbar==3.6.0
redis==4.3.4
```

```
django-redisboard==8.3.0
djangorestframework==3.13.1
requests==2.28.1
channels==3.0.5
channels-redis==3.4.1
psycopg2==2.9.3
uwsgi==2.0.20
daphne==3.0.2
```

In addition to the Python packages that you have installed in the previous chapters, the `requirements.txt` includes the following packages:

- `psycopg2`: A PostgreSQL adapter. You will use PostgreSQL for the production environment.
- `uwsgi`: A WSGI web server. You will configure this web server later to serve Django in the production environment.
- `daphne`: An ASGI web server. You will use this web server later to serve Django Channels.

Let's start by setting up the Docker application in Docker Compose. We will create a Docker Compose file with the definition for the web server, database, and Redis services.

Creating a Docker Compose file

To define the services that will run in different Docker containers, we will use a Docker Compose file. The Compose file is a text file with YAML format, defining services, networks, and data volumes for a Docker application. YAML is a human-readable data-serialization language. You can see an example of a YAML file at `https://yaml.org/`.

Next to the `educa` project directory, create a new file and name it `docker-compose.yml`. Add the following code to it:

```
services:

  web:
    build: .
    command: python /code/educa/manage.py runserver 0.0.0.0:8000
    restart: always
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    environment:
      - DJANGO_SETTINGS_MODULE=educa.settings.prod
```

In this file, you define a `web` service. The sections to define this service are as follows:

- `build`: Defines the build requirements for a service container image. This can be a single string defining a context path, or a detailed build definition. You provide a relative path with a single dot `.` to point to the same directory where the Compose file is located. Docker Compose will look for a `Dockerfile` at this location. You can read more about the `build` section at `https://docs.docker.com/compose/compose-file/build/`.
- `command`: Overrides the default command of the container. You run the Django development server using the `runserver` management command. The project is served on host `0.0.0.0`, which is the default Docker IP, on port `8000`.
- `restart`: Defines the restart policy for the container. Using `always`, the container is restarted always if it stops. This is useful for a production environment, where you want to minimize downtime. You can read more about the restart policy at `https://docs.docker.com/config/containers/start-containers-automatically/`.
- `volumes`: Data in Docker containers is not permanent. Each Docker container has a virtual filesystem that is populated with the files of the image and that is destroyed when the container is stopped. Volumes are the preferred method to persist data generated and used by Docker containers. In this section, you mount the local directory `.` to the `/code` directory of the image. You can read more about Docker volumes at `https://docs.docker.com/storage/volumes/`.
- `ports`: Exposes container ports. Host port `8000` is mapped to container port `8000`, on which the Django development server is running.
- `environment`: Defines environment variables. You set the `DJANGO_SETTINGS_MODULE` environment variable to use the production Django settings file `educa.settings.prod`.

Note that in the Docker Compose file definition, you are using the Django development server to serve the application. The Django development server is not suitable for production use, so you will replace it later with a WSGI Python web server.

You can find information about the Docker Compose specification at `https://docs.docker.com/compose/compose-file/`.

At this point, assuming your parent directory is named `Chapter17`, the file structure should look as follows:

```
Chapter17/
    Dockerfile
    docker-compose.yml
    educa/
        manage.py
        ...
    requirements.txt
```

Open a shell in the parent directory, where the `docker-compose.yml` file is located, and run the following command:

```
docker compose up
```

This will start the Docker app defined in the Docker Compose file. You will see an output that includes the following lines:

```
chapter17-web-1  | Performing system checks...
chapter17-web-1  |
chapter17-web-1  | System check identified no issues (0 silenced).
chapter17-web-1  | July 19, 2022 - 15:56:28
chapter17-web-1  | Django version 4.1, using settings 'educa.settings.prod'
chapter17-web-1  | Starting ASGI/Channels version 3.0.5 development server at
http://0.0.0.0:8000/
chapter17-web-1  | Quit the server with CONTROL-C.
```

The Docker container for your Django project is running!

Open `http://localhost:8000/admin/` with your browser. You should see the Django administration site login form. It should look like *Figure 17.2*:

Figure 17.2: The Django administration site login form

CSS styles are not being loaded. You are using `DEBUG=False`, so URL patterns for serving static files are not being included in the main `urls.py` file of the project. Remember that the Django development server is not suitable for serving static files. You will configure a server for serving static files later in this chapter.

If you access any other URL of your site, you might get an HTTP `500` error because you haven't configured a database for the production environment yet.

Take a look at the Docker Desktop app. You will see the following containers:

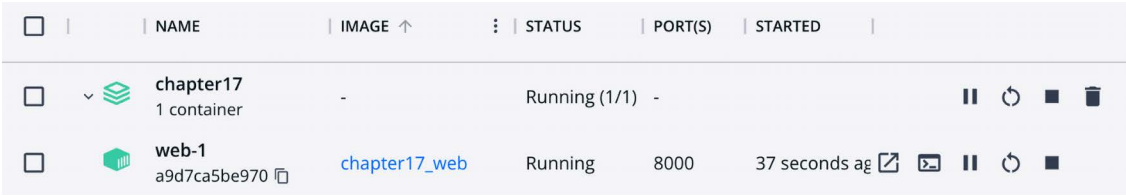


Figure 17.3: The chapter17 application and the web-1 container in Docker Desktop

The `chapter17` Docker application is running and it has a single container named `web-1`, which is running on port `8000`. The name for the Docker application is generated dynamically using the name of the directory where the Docker Compose file is located, in this case, `chapter17`.

Next, you are going to add a PostgreSQL service and a Redis service to your Docker application.

Configuring the PostgreSQL service

Throughout this book, you have mostly used the SQLite database. SQLite is simple and quick to set up, but for a production environment, you will need a more powerful database, such as PostgreSQL, MySQL, or Oracle. You learned how to install PostgreSQL in *Chapter 3*, *Extending Your Blog Application*. For the production environment, we will use a PostgreSQL Docker image instead. You can find information about the official PostgreSQL Docker image at `https://hub.docker.com/_/postgres`.

Edit the `docker-compose.yml` file and add the following lines highlighted in bold:

```
services:

  db:
    image: postgres:14.5
    restart: always
    volumes:
      - ./data/db:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres

  web:
    build: .
    command: python /code/educa/manage.py runserver 0.0.0.0:8000
```

```
    restart: always
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    environment:
      - DJANGO_SETTINGS_MODULE=educa.settings.prod
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
      - db
```

With these changes, you define a service named `db` with the following subsections:

- `image`: The service uses the base `postgres` Docker image.
- `restart`: The restart policy is set to `always`.
- `volumes`: You mount the `./data/db` directory to the image directory `/var/lib/postgresql/data` to persist the database so that data stored in the database is maintained after the Docker application is stopped. This will create the local `data/db/` path.
- `environment`: You use the `POSTGRES_DB` (database name), `POSTGRES_USER`, and `POSTGRES_PASSWORD` variables with default values.

The definition for the `web` service now includes the PostgreSQL environment variables for Django. You create a service dependency using `depends_on` so that the `web` service is started after the `db` service. This will guarantee the order of the container initialization, but it won't guarantee that PostgreSQL is fully initiated before the Django web server is started. To solve this, you need to use a script that will wait on the availability of the database host and its TCP port. Docker recommends using the `wait-for-it` tool to control container initialization.

Download the `wait-for-it.sh` Bash script from `https://github.com/vishnubob/wait-for-it/blob/master/wait-for-it.sh` and save the file next to the `docker-compose.yml` file. Then edit the `docker-compose.yml` file and modify the `web` service definition as follows. New code is highlighted in bold:

```
web:
  build: .
  command: ["./wait-for-it.sh", "db:5432", "--",
            "python", "/code/educa/manage.py", "runserver",
            "0.0.0.0:8000"]
  restart: always
  volumes:
      - .:/code
    environment:
```

```
      - DJANGO_SETTINGS_MODULE=educa.settings.prod
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
      - db
```

In this service definition, you use the `wait-for-it.sh` Bash script to wait for the `db` host to be ready and accepting connections on port `5432`, the default port for PostgreSQL, before starting the Django development server. You can read more about the service startup order in Compose at `https://docs.docker.com/compose/startup-order/`.

Let's edit Django settings. Edit the `educa/settings/prod.py` file and add the following code highlighted in bold:

```
import os
from .base import *

DEBUG = False

ADMINS = [
    ('Antonio M', 'email@mydomain.com'),
]

ALLOWED_HOSTS = ['*']

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB'),
        'USER': os.environ.get('POSTGRES_USER'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD'),
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

In the production settings file, you use the following settings:

- `ENGINE`: You use the Django database backend for PostgreSQL.
- `NAME`, `USER`, and `PASSWORD`: You use `os.environ.get()` to retrieve the environment variables `POSTGRES_DB` (database name), `POSTGRES_USER`, and `POSTGRES_PASSWORD`. You have set these environment variables in the Docker Compose file.

- `HOST`: You use `db`, which is the container hostname for the database service defined in the Docker Compose file. A container hostname defaults to the container's ID in Docker. That's why you use the `db` hostname.
- `PORT`: You use the value `5432`, which is the default port for PostgreSQL.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

The first execution after adding the `db` service to the Docker Compose file will take longer because PostgreSQL needs to initialize the database. The output will contain the following two lines:

```
chapter17-db-1   | database system is ready to accept connections
...
chapter17-web-1  | Starting ASGI/Channels version 3.0.5 development server at
http://0.0.0.0:8000/
```

Both the PostgreSQL database and the Django application are ready. The production database is empty, so you need to apply database migrations.

Applying database migrations and creating a superuser

Open a different shell in the parent directory, where the `docker-compose.yml` file is located, and run the following command:

```
docker compose exec web python /code/educa/manage.py migrate
```

The command `docker compose exec` allows you to execute commands in the container. You use this command to execute the `migrate` management command in the `web` Docker container.

Finally, create a superuser with the following command:

```
docker compose exec web python /code/educa/manage.py createsuperuser
```

Migrations have been applied to the database and you have created a superuser. You can access `http://localhost:8000/admin/` with the superuser credentials. CSS styles still won't load because you haven't configured serving static files yet.

You have defined services to serve Django and PostgreSQL using Docker Compose. Next, you will add a service to serve Redis in the production environment.

Configuring the Redis service

Let's add a Redis service to the Docker Compose file. For this purpose, you will use the official Redis Docker image. You can find information about the official Redis Docker image at `https://hub.docker.com/_/redis`.

Edit the `docker-compose.yml` file and add the following lines highlighted in bold:

```
services:

  db:
    # ...

  cache:
    image: redis:7.0.4
    restart: always
    volumes:
      - ./data/cache:/data

  web:
    # ...
    depends_on:
      - db
      - cache
```

In the previous code, you define the `cache` service with the following subsections:

- `image`: The service uses the base `redis` Docker image.
- `restart`: The restart policy is set to `always`.
- `volumes`: You mount the `./data/cache` directory to the image directory `/data` where any Redis writes will be persisted. This will create the local `data/cache/` path.

In the `web` service definition, you add the `cache` service as a dependency, so that the `web` service is started after the `cache` service. The Redis server initializes fast, so you don't need to use the `wait-for-it` tool in this case.

Edit the `educa/settings/prod.py` file and add the following lines:

```
REDIS_URL = 'redis://cache:6379'
CACHES['default']['LOCATION'] = REDIS_URL
CHANNEL_LAYERS['default']['CONFIG']['hosts'] = [REDIS_URL]
```

In these settings, you use the `cache` hostname that is automatically generated by Docker Compose using the name of the `cache` service and port `6379` used by Redis. You modify the Django `CACHE` setting and the `CHANNEL_LAYERS` setting used by Channels to use the production Redis URL.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Open the Docker Desktop application. You should see now the `chapter17` Docker application running a container for each service defined in the Docker Compose file: `db`, `cache`, and `web`:

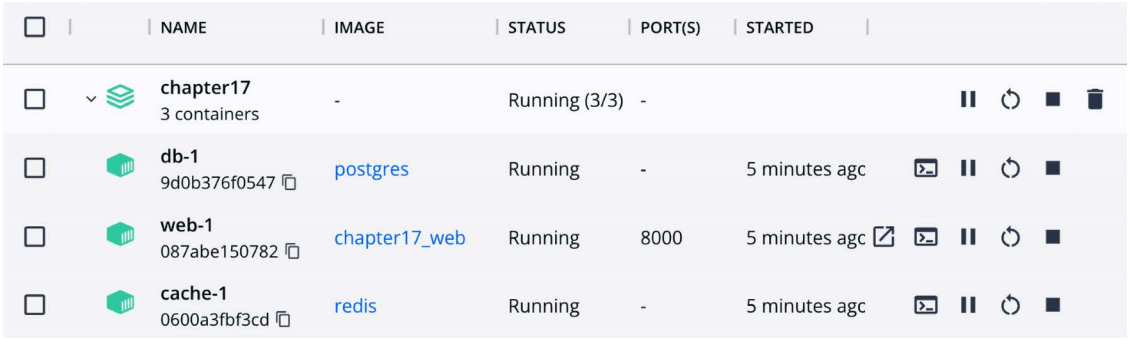


Figure 17.4: The chapter17 application with the db-1, web-1, and cache-1 containers in Docker Desktop

You are still serving Django with the Django development server, which is not suitable for production use. Let's replace it with the WSGI Python web server.

Serving Django through WSGI and NGINX

Django's primary deployment platform is WSGI. **WSGI** stands for **Web Server Gateway Interface**, and it is the standard for serving Python applications on the web.

When you generate a new project using the `startproject` command, Django creates a `wsgi.py` file inside your project directory. This file contains a WSGI application callable, which is an access point to your application.

WSGI is used for both running your project with the Django development server and deploying your application with the server of your choice in a production environment. You can learn more about WSGI at `https://wsgi.readthedocs.io/en/latest/`.

Using uWSGI

Throughout this book, you have been using the Django development server to run projects in your local environment. However, you need a standard web server for deploying your application in a production environment.

uWSGI is an extremely fast Python application server. It communicates with your Python application using the WSGI specification. uWSGI translates web requests into a format that your Django project can process.

Let's configure uWSGI to serve the Django project. You already added `uwsgi==2.0.20` to the `requirements.txt` file of the project, so uWSGI is already being installed in the Docker image of the `web` service.

Edit the `docker-compose.yml` file and modify the `web` service definition as follows. New code is highlighted in bold:

```
web:
    build: .
    command: ["./wait-for-it.sh", "db:5432", "--",
              "uwsgi", "--ini", "/code/config/uwsgi/uwsgi.ini"]
    restart: always
    volumes:
      - .:/code
    environment:
      - DJANGO_SETTINGS_MODULE=educa.settings.prod
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
      - db
      - cache
```

Make sure to remove the `ports` section. uWSGI will be reachable with a socket, so you don't need to expose a port in the container.

The new `command` for the image runs `uwsgi` passing the configuration file `/code/config/uwsgi/uwsgi.ini` to it. Let's create the configuration file for uWSGI.

Configuring uWSGI

uWSGI allows you to define a custom configuration in a `.ini` file. Next to the `docker-compose.yml` file, create the file path `config/uwsgi/uwsgi.ini`. Assuming your parent directory is named `Chapter17`, the file structure should look as follows:

```
Chapter17/
    config/
        uwsgi/
            uwsgi.ini
    Dockerfile
    docker-compose.yml
    educa/
        manage.py
        ...
    requirements.txt
```

Edit the `config/uwsgi/uwsgi.ini` file and add the following code to it:

```
[uwsgi]
socket=/code/educa/uwsgi_app.sock
chdir = /code/educa/
module=educa.wsgi:application
master=true
chmod-socket=666
uid=www-data
gid=www-data
vacuum=true
```

In the `uwsgi.ini` file, you define the following options:

- `socket`: The UNIX/TCP socket to bind the server.
- `chdir`: The path to your project directory, so that uWSGI changes to that directory before loading the Python application.
- `module`: The WSGI module to use. You set this to the `application` callable contained in the `wsgi` module of your project.
- `master`: Enable the master process.
- `chmod-socket`: The file permissions to apply to the socket file. In this case, you use `666` so that NGINX can read/write the socket.
- `uid`: The user ID of the process once it's started.
- `gid`: The group ID of the process once it's started.
- `vacuum`: Using `true` instructs uWSGI to clean up any temporary files or UNIX sockets it creates.

The `socket` option is intended for communication with some third-party router, such as NGINX. You are going to run uWSGI using a socket and you are going to configure NGINX as your web server, which will communicate with uWSGI through the socket.

You can find the list of available uWSGI options at `https://uwsgi-docs.readthedocs.io/en/latest/Options.html`.

You will not be able to access your uWSGI instance from your browser now, since it's running through a socket. Let's complete the production environment.

Using NGINX

When you are serving a website, you have to serve dynamic content, but you also need to serve static files, such as CSS style sheets, JavaScript files, and images. While uWSGI is capable of serving static files, it adds an unnecessary overhead to HTTP requests and therefore, it is encouraged to set up a web server, such as NGINX, in front of it.

NGINX is a web server focused on high concurrency, performance, and low memory usage. NGINX also acts as a reverse proxy, receiving HTTP and WebSocket requests and routing them to different backends.

Generally, you will use a web server, such as NGINX, in front of uWSGI for serving static files efficiently, and you will forward dynamic requests to uWSGI workers. By using NGINX, you can also apply different rules and benefit from its reverse proxy capabilities.

We will add the NGINX service to the Docker Compose file using the official NGINX Docker image. You can find information about the official NGINX Docker image at `https://hub.docker.com/_/nginx`.

Edit the `docker-compose.yml` file and add the following lines highlighted in bold:

```
services:

  db:
    # ...

  cache:
    # ...

  web:
    # ...

  nginx:
    image: nginx:1.23.1
    restart: always
    volumes:
      - ./config/nginx:/etc/nginx/templates
      - .:/code
    ports:
      - "80:80"
```

You have added the definition for the `nginx` service with the following subsections:

- `image`: The service uses the base `nginx` Docker image.
- `restart`: The restart policy is set to `always`.
- `volumes`: You mount the `./config/nginx` volume to the `/etc/nginx/templates` directory of the Docker image. This is where NGINX will look for a default configuration template. You also mount the local directory `.` to the `/code` directory of the image, so that NGINX can have access to static files.
- `ports`: You expose port `80`, which is mapped to container port `80`. This is the default port for HTTP.

Let's configure the NGINX web server.

Configuring NGINX

Create the following file path highlighted in bold under the `config/` directory:

```
config/
    uwsgi/
      uwsgi.ini
    nginx/
        default.conf.template
```

Edit the file `nginx/default.conf.template` and add the following code to it:

```
# upstream for uWSGI
upstream uwsgi_app {
    server unix:/code/educa/uwsgi_app.sock;
}

server {
    listen       80;
    server_name  www.educaproject.com educaproject.com;
    error_log    stderr warn;
    access_log   /dev/stdout main;

    location / {
        include      /etc/nginx/uwsgi_params;
        uwsgi_pass   uwsgi_app;
    }
}
```

This is the basic configuration for NGINX. In this configuration, you set up an upstream named `uwsgi_app`, which points to the socket created by uWSGI. You use the `server` block with the following configuration:

- You tell NGINX to listen on port `80`.
- You set the server name to both `www.educaproject.com` and `educaproject.com`. NGINX will serve incoming requests for both domains.
- You use `stderr` for the `error_log` directive to get error logs written to the standard error file. The second parameter determines the logging level. You use `warn` to get warnings and errors of higher severity.
- You point `access_log` to the standard output with `/dev/stdout`.
- You specify that any request under the `/` path has to be routed to the `uwsgi_app` socket to uWSGI.
- You include the default uWSGI configuration parameters that come with NGINX. These are located at `/etc/nginx/uwsgi_params`.

NGINX is now configured. You can find the NGINX documentation at `https://nginx.org/en/docs/`.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Open the URL `http://localhost/` in your browser. It's not necessary to add a port to the URL because you are accessing the host through the standard HTTP port `80`. You should see the course list page with no CSS styles, like *Figure 17.5*:

Figure 17.5: The course list page served with NGINX and uWSGI

The following diagram shows the request/response cycle of the production environment that you have set up:

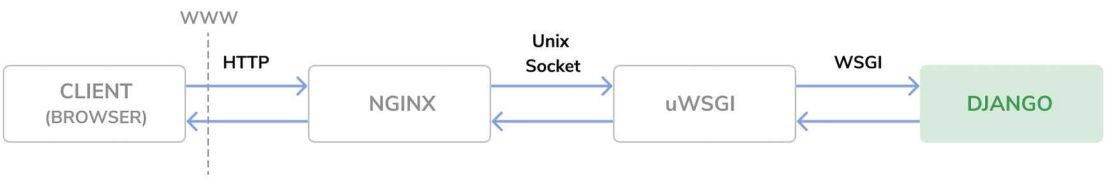


Figure 17.6: The production environment request/response cycle

The following happens when the client browser sends an HTTP request:

1. NGINX receives the HTTP request.
2. NGINX delegates the request to uWSGI through a socket.
3. uWSGI passes the request to Django for processing.
4. Django returns an HTTP response that is passed back to NGINX, which in turn passes it back to the client browser.

If you check the Docker Desktop application, you should see that there are 4 containers running:

- `db` service running PostgreSQL
- `cache` service running Redis
- `web` service running uWSGI + Django
- `nginx` service running NGINX

Let's continue with the production environment setup. Instead of accessing our project using `localhost`, we will configure the project to use the `educaproject.com` hostname.

Using a hostname

You will use the `educaproject.com` hostname for your site. Since you are using a sample domain name, you need to redirect it to your local host.

If you are using Linux or macOS, edit the `/etc/hosts` file and add the following line to it:

```
127.0.0.1 educaproject.com www.educaproject.com
```

If you are using Windows, edit the file `C:\Windows\System32\drivers\etc` and add the same line.

By doing so, you are routing the hostnames `educaproject.com` and `www.educaproject.com` to your local server. In a production server, you won't need to do this, since you will have a fixed IP address and you will point your hostname to your server in your domain's DNS configuration.

Open `http://educaproject.com/` in your browser. You should be able to see your site, still without any static assets loaded. Your production environment is almost ready.

Now you can restrict the hosts that can serve your Django project. Edit the production settings file `educa/settings/prod.py` of your project and change the `ALLOWED_HOSTS` setting, as follows:

```
ALLOWED_HOSTS = ['educaproject.com', 'www.educaproject.com']
```

Django will only serve your application if it's running under any of these hostnames. You can read more about the `ALLOWED_HOSTS` setting at `https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts`.

The production environment is almost ready. Let's continue by configuring NGINX to serve static files.

Serving static and media assets

uWSGI is capable of serving static files flawlessly, but it is not as fast and effective as NGINX. For the best performance, you will use NGINX to serve static files in your production environment. You will set up NGINX to serve both the static files of your application (CSS style sheets, JavaScript files, and images) and media files uploaded by instructors for the course contents.

Edit the `settings/base.py` file and add the following line just below the `STATIC_URL` setting:

```
STATIC_ROOT = BASE_DIR / 'static'
```

This is the root directory for all static files of the project. Next, you are going to collect the static files from the different Django applications into the common directory.

Collecting static files

Each application in your Django project may contain static files in a `static/` directory. Django provides a command to collect static files from all applications into a single location. This simplifies the setup for serving static files in production. The `collectstatic` command collects the static files from all applications of the project into the path defined with the `STATIC_ROOT` setting.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Open another shell in the parent directory, where the `docker-compose.yml` file is located, and run the following command:

```
docker compose exec web python /code/educa/manage.py collectstatic
```

Note that you can alternatively run the following command in the shell, from the `educa/` project directory:

```
python manage.py collectstatic --settings=educa.settings.local
```

Both commands will have the same effect since the base local directory is mounted to the Docker image. Django will ask if you want to override any existing files in the root directory. Type `yes` and press `Enter`. You will see the following output:

```
171 static files copied to '/code/educa/static'.
```

Files located under the `static/` directory of each application present in the `INSTALLED_APPS` setting have been copied to the global `/educa/static/` project directory.

Serving static files with NGINX

Edit the `config/nginx/default.conf.template` file and add the following lines highlighted in bold to the `server` block:

```
server {
    # ...

    location / {
        include      /etc/nginx/uwsgi_params;
        uwsgi_pass   uwsgi_app;
    }

    location /static/ {
        alias /code/educa/static/;
```

```
    }
    location /media/ {
        alias /code/educa/media/;
    }
}
```

These directives tell NGINX to serve static files located under the `/static/` and `/media/` paths directly. These paths are as follows:

- `/static/`: Corresponds to the path of the `STATIC_URL` setting. The target path corresponds to the value of the `STATIC_ROOT` setting. You use it to serve the static files of your application from the directory mounted to the NGINX Docker image.
- `/media/`: Corresponds to the path of the `MEDIA_URL` setting, and its target path corresponds to the value of the `MEDIA_ROOT` setting. You use it to serve the media files uploaded to the course contents from the directory mounted to the NGINX Docker image.

The schema of the production environment now looks like this:

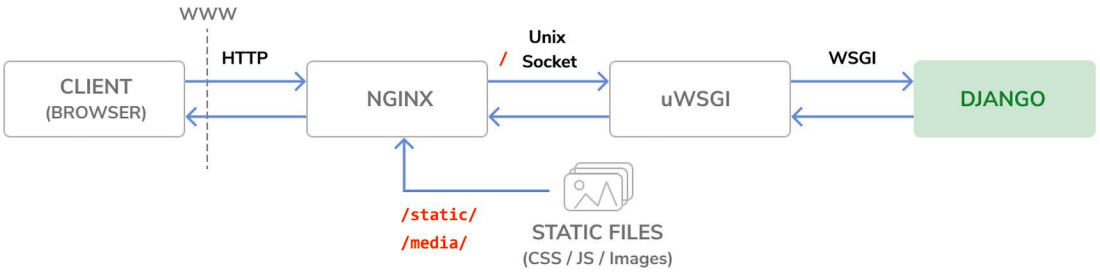


Figure 17.7: The production environment request/response cycle, including static files

Files under the `/static/` and `/media/` paths are now served by NGINX directly, instead of being forwarded to uWSGI. Requests to any other path are still passed by NGINX to uWSGI through the UNIX socket.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Open `http://educaproject.com/` in your browser. You should see the following screen:

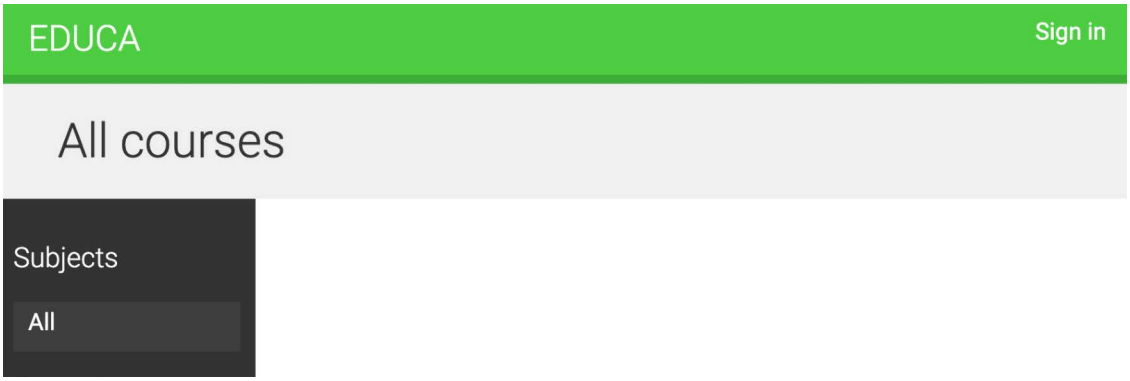


Figure 17.8: The course list page served with NGINX and uWSGI

Static resources, such as CSS style sheets and images, are now loaded correctly. HTTP requests for static files are now being served by NGINX directly, instead of being forwarded to uWSGI.

You have successfully configured NGINX for serving static files. Next, you are going to check your Django project to deploy it in a production environment and you are going to serve your site under HTTPS.

Securing your site with SSL/TLS

The **Transport Layer Security** (**TLS**) protocol is the standard for serving websites through a secure connection. The TLS predecessor is **Secure Sockets Layer** (**SSL**). Although SSL is now deprecated, in multiple libraries and online documentation, you will find references to both the terms TLS and SSL. It's strongly encouraged that you serve your websites over HTTPS.

In this section, you are going to check your Django project for a production deployment and prepare the project to be served over HTTPS. Then, you are going to configure an SSL/TLS certificate in NGINX to serve your site securely.

Checking your project for production

Django includes a system check framework for validating your project at any time. The check framework inspects the applications installed in your Django project and detects common problems. Checks are triggered implicitly when you run management commands like `runserver` and `migrate`. However, you can trigger checks explicitly with the `check` management command.

You can read more about Django's system check framework at `https://docs.djangoproject.com/en/4.1/topics/checks/`.

Let's confirm that the check framework does not raise any issues for your project. Open the shell in the `educa` project directory and run the following command to check your project:

```
python manage.py check --settings=educa.settings.prod
```

You will see the following output:

```
System check identified no issues (0 silenced).
```

The system check framework didn't identify any issues. If you use the `--deploy` option, the system check framework will perform additional checks that are relevant for a production deployment.

Run the following command from the `educa` project directory:

```
python manage.py check --deploy --settings=educa.settings.prod
```

You will see output like the following:

```
System check identified some issues:

WARNINGS:
(security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting.
...
(security.W008) Your SECURE_SSL_REDIRECT setting is not set to True...
(security.W009) Your SECRET_KEY has less than 50 characters, less than 5 unique
characters, or it's prefixed with 'django-insecure-'...
(security.W012) SESSION_COOKIE_SECURE is not set to True. ...
(security.W016) You have 'django.middleware.csrf.CsrfViewMiddleware' in your
MIDDLEWARE, but you have not set CSRF_COOKIE_SECURE ...

System check identified 5 issues (0 silenced).
```

The check framework has identified five issues (0 errors, 5 warnings). All warnings are related to security-related settings.

Let's address issue `security.W009`. Edit the `educa/settings/base.py` file and modify the `SECRET_KEY` setting by removing the `django-insecure-` prefix and adding additional random characters to generate a string with at least 50 characters.

Run the `check` command again and verify that issue `security.W009` is not raised anymore. The rest of the warnings are related to SSL/TLS configuration. We will address them next.

Configuring your Django project for SSL/TLS

Django comes with specific settings for SSL/TLS support. You are going to edit the production settings to serve your site over HTTPS.

Edit the `educa/settings/prod.py` settings file and add the following settings to it:

```
# Security
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
SECURE_SSL_REDIRECT = True
```

These settings are as follows:

- `CSRF_COOKIE_SECURE`: Use a secure cookie for **cross-site request forgery** (**CSRF**) protection. With `True`, browsers will only transfer the cookie over HTTPS.
- `SESSION_COOKIE_SECURE`: Use a secure session cookie. With `True`, browsers will only transfer the cookie over HTTPS.
- `SECURE_SSL_REDIRECT`: Whether HTTP requests have to be redirected to HTTPS.

Django will now redirect HTTP requests to HTTPS; session and CSRF cookies will be sent only over HTTPS.

Run the following command from the main directory of your project:

```
python manage.py check --deploy --settings=educa.settings.prod
```

Only one warning remains, `security.W004`:

```
(security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting.
...
```

This warning is related to the **HTTP Strict Transport Security** (**HSTS**) policy. The HSTS policy prevents users from bypassing warnings and connecting to a site with an expired, self-signed, or otherwise invalid SSL certificate. In the next section, we will use a self-signed certificate for our site, so we will ignore this warning. When you own a real domain, you can apply for a trusted **Certificate Authority** (**CA**) to issue an SSL/TLS certificate for it, so that browsers can verify its identity. In that case, you can give a value to `SECURE_HSTS_SECONDS` higher than `0`, which is the default value. You can learn more about the HSTS policy at `https://docs.djangoproject.com/en/4.1/ref/middleware/#http-strict-transport-security`.

You have successfully fixed the rest of the issues raised by the check framework. You can read more about the Django deployment checklist at `https://docs.djangoproject.com/en/4.1/howto/deployment/checklist/`.

Creating an SSL/TLS certificate

Create a new directory inside the `educa` project directory and name it `ssl`. Then, generate an SSL/TLS certificate from the command line with the following command:

```
openssl req -x509 -newkey rsa:2048 -sha256 -days 3650 -nodes \
  -keyout ssl/educa.key -out ssl/educa.crt \
  -subj '/CN=*.educaproject.com' \
  -addext 'subjectAltName=DNS:*.educaproject.com'
```

This will generate a private key and a 2048-bit SSL/TLS certificate that is valid for 10 years. This certificate is issued for the hostname `*.educaproject.com`. This is a wildcard certificate; by using the wildcard character `*` in the domain name, the certificate can be used for any subdomain of `educaproject.com`, such as `www.educaproject.com` or `django.educaproject.com`. After generating the certificate, the `educa/ssl/` directory will contain two files: `educa.key` (the private key) and `educa.crt` (the certificate).

You will need at least OpenSSL 1.1.1 or LibreSSL 3.1.0 to use the `-addext` option. You can check the OpenSSL location in your machine with the command `which openssl` and you can check the version with the command `openssl version`.

Alternatively, you can use the SSL/TLS certificate provided in the source code for this chapter. You will find the certificate at `https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/educa/ssl/`. Note that you should generate a private key and not use this certificate in production.

Configuring NGINX to use SSL/TLS

Edit the `docker-compose.yml` file and add the following line highlighted in bold:

```
services:
  # ...

  nginx:
    #...
    ports:
      - "80:80"
      - "443:443"
```

The NGINX container host will be accessible through port `80` (HTTP) and port `443` (HTTPS). The host port `443` is mapped to the container port `443`.

Edit the `config/nginx/default.conf.template` file of the `educa` project and edit the `server` block to include SSL/TLS, as follows:

```
server {
    listen              80;
    listen              443 ssl;
    ssl_certificate     /code/educa/ssl/educa.crt;
    ssl_certificate_key /code/educa/ssl/educa.key;
    server_name         www.educaproject.com educaproject.com;
    # ...
}
```

With the preceding code, NGINX now listens both to HTTP over port `80` and HTTPS over port `443`. You indicate the path to the SSL/TLS certificate with `ssl_certificate` and the certificate key with `ssl_certificate_key`.

Stop the Docker application from the shell by pressing the keys *Ctrl* + *C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Open `https://educaproject.com/` with your browser. You should see a warning message similar to the following one:

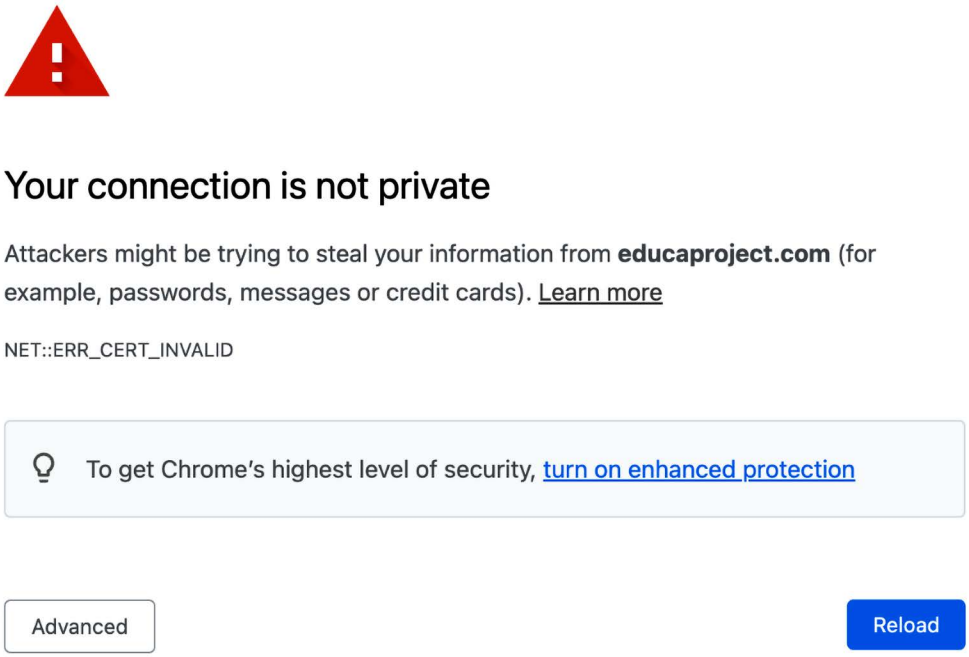


Figure 17.9: An invalid certificate warning

This screen might vary depending on your browser. It alerts you that your site is not using a trusted or valid certificate; the browser can't verify the identity of your site. This is because you signed your own certificate instead of obtaining one from a trusted CA. When you own a real domain, you can apply for a trusted CA to issue an SSL/TLS certificate for it, so that browsers can verify its identity. If you want to obtain a trusted certificate for a real domain, you can refer to the Let's Encrypt project created by the Linux Foundation. It is a nonprofit CA that simplifies obtaining and renewing trusted SSL/TLS certificates for free. You can find more information at `https://letsencrypt.org`.

Click on the link or button that provides additional information and choose to visit the website, ignoring warnings. The browser might ask you to add an exception for this certificate or verify that you trust it. If you are using Chrome, you might not see any option to proceed to the website. If this is the case, type `thisisunsafe` and press `Enter` directly in Chrome on the warning page. Chrome will then load the website. Note that you do this with your own issued certificate; don't trust any unknown certificate or bypass the browser SSL/TLS certificate checks for other domains.

When you access the site, the browser will display a lock icon next to the URL like *Figure 17.10*:

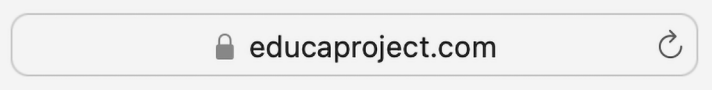


Figure 17.10: The browser address bar, including a secure connection padlock icon

Other browsers might display a warning indicating that the certificate is not trusted, like *Figure 17.11*:

Figure 17.11: The browser address bar, including a warning message

If you click the lock icon or the warning icon, the SSL/TLS certificate details will be displayed as follows:

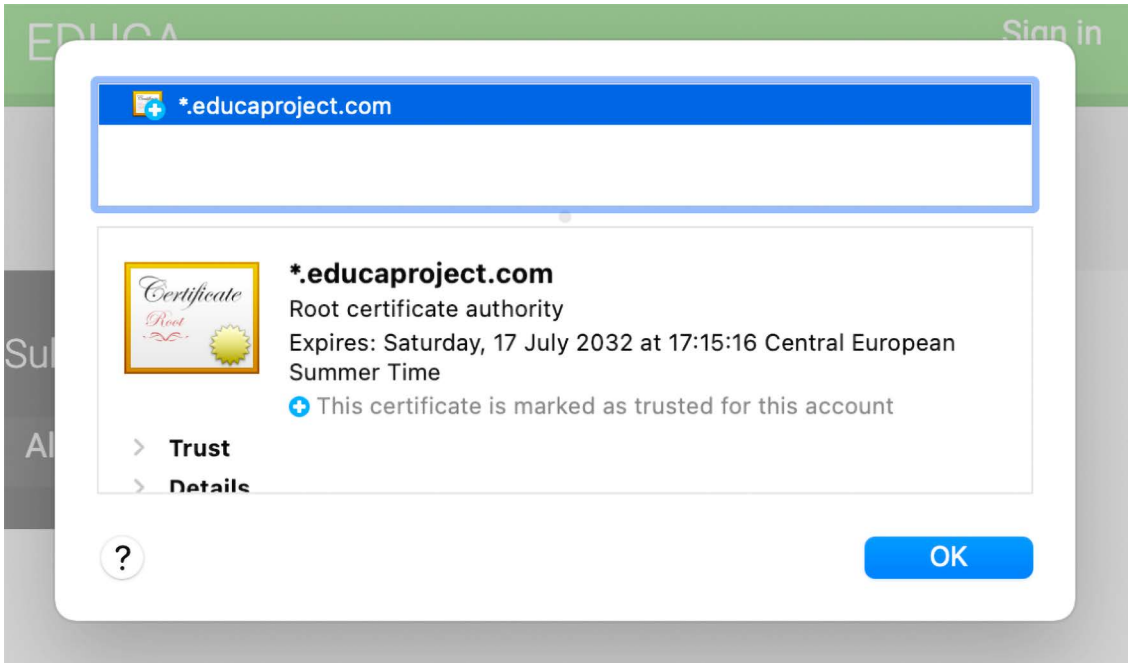


Figure 17.12: TLS/SSL certificate details

In the certificate details, you will see it is a self-signed certificate and you will see its expiration date. Your browser might mark the certificate as unsafe, but you are using it for testing purposes only. You are now serving your site securely over HTTPS.

Redirecting HTTP traffic over to HTTPS

You are redirecting HTTP requests to HTTPS with Django using the `SECURE_SSL_REDIRECT` setting. Any request using `http://` is redirected to the same URL using `https://`. However, this can be handled in a more efficient manner using NGINX.

Edit the `config/nginx/default.conf.template` file and add the following lines highlighted in bold:

```
# upstream for uWSGI
upstream uwsgi_app {
    server unix:/code/educa/uwsgi_app.sock;
}

server {
    listen      80;
    server_name www.educaproject.com educaproject.com;
    return 301 https://$host$request_uri;
}

server {
    listen               443 ssl;
    ssl_certificate      /code/educa/ssl/educa.crt;
    ssl_certificate_key  /code/educa/ssl/educa.key;
    server_name   www.educaproject.com educaproject.com;
    # ...
}
```

In this code, you remove the directive `listen 80;` from the original `server` block, so that the platform is only available over HTTPS (port `443`). On top of the original `server` block, you add an additional `server` block that only listens on port `80` and redirects all HTTP requests to HTTPS. To achieve this, you return an HTTP response code `301` (permanent redirect) that redirects to the `https://` version of the requested URL using the `$host` and `$request_uri` variables.

Open a shell in the parent directory, where the `docker-compose.yml` file is located, and run the following command to reload NGINX:

```
docker compose exec nginx nginx -s reload
```

This runs the `nginx -s reload` command in the `nginx` container. You are now redirecting all HTTP traffic to HTTPS using NGINX.

Your environment is now secured with TLS/SSL. To complete the production environment, you need to set up an asynchronous web server for Django Channels.

Using Daphne for Django Channels

In *Chapter 16*, *Building a Chat Server*, you used Django Channels to build a chat server using WebSockets. uWSGI is suitable for running Django or any other WSGI application, but it doesn't support asynchronous communication using **Asynchronous Server Gateway Interface** (**ASGI**) or WebSockets. In order to run Channels in production, you need an ASGI web server that is capable of managing WebSockets.

Daphne is an HTTP, HTTP2, and WebSocket server for ASGI developed to serve Channels. You can run Daphne alongside uWSGI to serve both ASGI and WSGI applications efficiently. You can find more information about Daphne at `https://github.com/django/daphne`.

You already added `daphne==3.0.2` to the `requirements.txt` file of the project. Let's create a new service in the Docker Compose file to run the Daphne web server.

Edit the `docker-compose.yml` file and add the following lines:

```
daphne:
    build: .
    working_dir: /code/educa/
    command: ["../wait-for-it.sh", "db:5432", "--",
              "daphne", "-u", "/code/educa/daphne.sock",
              "educa.asgi:application"]
    restart: always
    volumes:
      - .:/code
    environment:
      - DJANGO_SETTINGS_MODULE=educa.settings.prod
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
      - db
      - cache
```

The `daphne` service definition is very similar to the `web` service. The image for the `daphne` service is also built with the `Dockerfile` you previously created for the `web` service. The main differences are:

- `working_dir` changes the working directory of the image to `/code/educa/`.
- `command` runs the `educa.asgi:application` application defined in the `educa/asgi.py` file with `daphne` using a UNIX socket. It also uses the `wait-for-it` Bash script to wait for the PostgreSQL database to be ready before initializing the web server.

Since you are running Django on production, Django checks the `ALLOWED_HOSTS` when receiving HTTP requests. We will implement the same validation for WebSocket connections.

Edit the `educa/asgi.py` file of your project and add the following lines highlighted in bold:

```
import os

from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from channels.auth import AuthMiddlewareStack
import chat.routing

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
    'http': django_asgi_app,
    'websocket': AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter(chat.routing.websocket_urlpatterns)
        )
    ),
})
```

The Channels configuration is now ready for production.

Using secure connections for WebSockets

You have configured NGINX to use secure connections with SSL/TLS. You need to change `ws` (WebSocket) connections to use the `wss` (WebSocket Secure) protocol now, in the same way that HTTP connections are now being served over HTTPS.

Edit the `chat/room.html` template of the `chat` application and find the following line in the `domready` block:

```
const url = 'ws://' + window.location.host +
```

Replace that line with the following one:

```
const url = 'wss://' + window.location.host +
```

By using `wss://` instead of `ws://`, you are explicitly connecting to a secure WebSocket.

Including Daphne in the NGINX configuration

In your production setup, you will run Daphne on a UNIX socket and use NGINX in front of it. NGINX will pass requests to Daphne based on the requested path. You will expose Daphne to NGINX through a UNIX socket interface, just like the uWSGI setup.

Edit the `config/nginx/default.conf.template` file and make it look as follows:

```
# upstream for uWSGI
upstream uwsgi_app {
    server unix:/code/educa/uwsgi_app.sock;
}

# upstream for Daphne
upstream daphne {
    server unix:/code/educa/daphne.sock;
}

server {
    listen       80;
    server_name www.educaproject.com educaproject.com;
    return 301 https://$host$request_uri;
}

server {
    listen               443 ssl;
    ssl_certificate      /code/educa/ssl/educa.crt;
    ssl_certificate_key  /code/educa/ssl/educa.key;
    server_name  www.educaproject.com educaproject.com;
    error_log    stderr warn;
    access_log   /dev/stdout main;

    location / {
        include      /etc/nginx/uwsgi_params;
        uwsgi_pass   uwsgi_app;
    }

    location /ws/ {
        proxy_http_version  1.1;
        proxy_set_header    Upgrade $http_upgrade;
        proxy_set_header    Connection "upgrade";
        proxy_redirect      off;
```

```
        proxy_pass          http://daphne;
    }

    location /static/ {
        alias /code/educa/static/;
    }
    location /media/ {
        alias /code/educa/media/;
    }

}
```

In this configuration, you set up a new upstream named `daphne`, which points to a UNIX socket created by Daphne. In the `server` block, you configure the `/ws/` location to forward requests to Daphne. You use the `proxy_pass` directive to pass requests to Daphne and you include some additional proxy directives.

With this configuration, NGINX will pass any URL request that starts with the `/ws/` prefix to Daphne and the rest to uWSGI, except for files under the `/static/` or `/media/` paths, which will be served directly by NGINX.

The production setup including Daphne now looks like this:

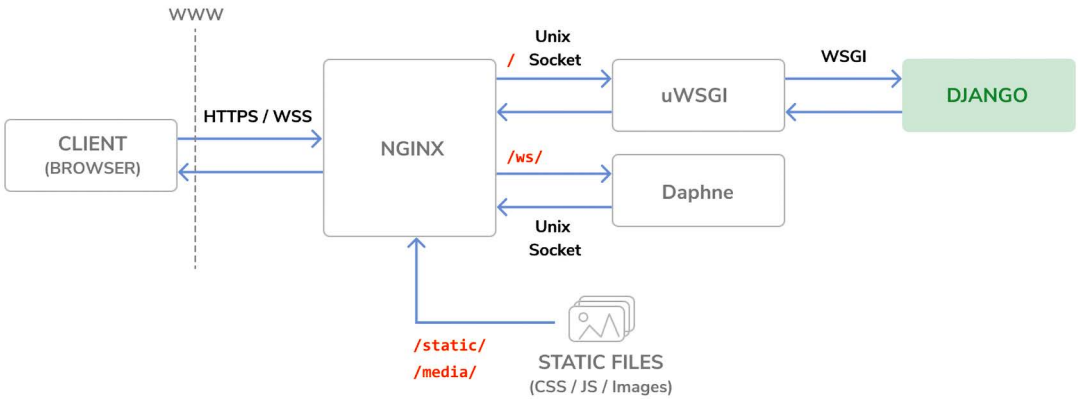


Figure 17.13: The production environment request/response cycle, including Daphne

NGINX runs in front of uWSGI and Daphne as a reverse proxy server. NGINX faces the web and passes requests to the application server (uWSGI or Daphne) based on their path prefix. Besides this, NGINX also serves static files and redirects non-secure requests to secure ones. This setup reduces downtime, consumes less server resources, and provides greater performance and security.

Stop the Docker application from the shell by pressing the keys *Ctrl + C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Use your browser to create a sample course with an instructor user, log in with a user who is enrolled on the course, and open `https://educaproject.com/chat/room/1/` with your browser. You should be able to send and receive messages like the following example:

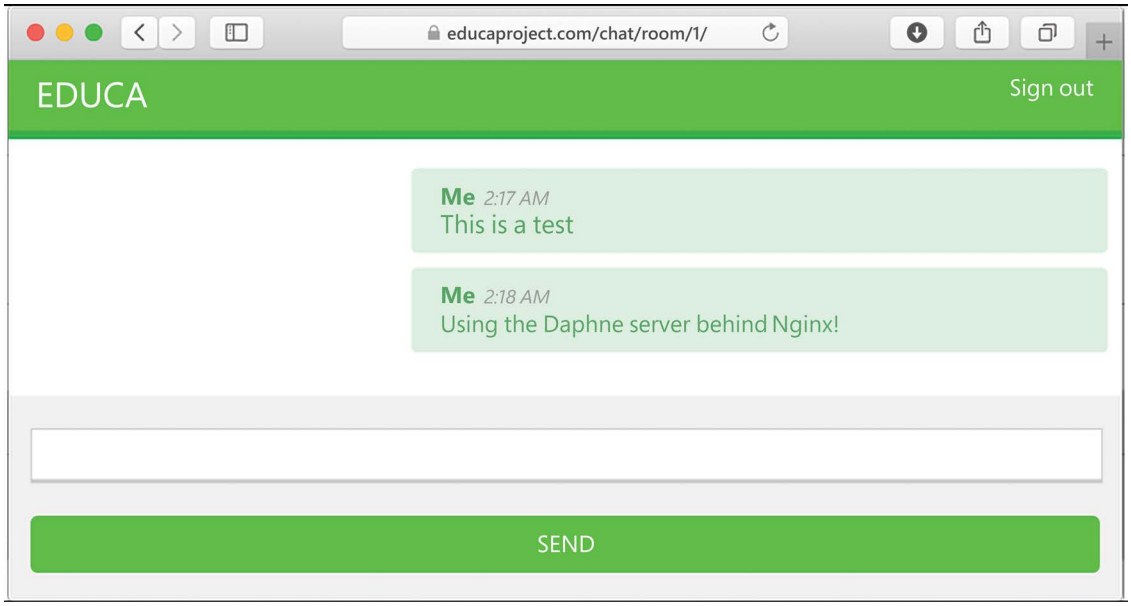


Figure 17.14: Course chat room messages served with NGINX and Daphne

Daphne is working correctly, and NGINX is passing WebSocket requests to it. All connections are secured with SSL/TLS.

Congratulations! You have built a custom production-ready stack using NGINX, uWSGI, and Daphne. You could do further optimization for additional performance and enhanced security through configuration settings in NGINX, uWSGI, and Daphne. However, this production setup is a great start!

You have used Docker Compose to define and run services in multiple containers. Note that you can use Docker Compose both for local development environments as well as production environments. You can find additional information on using Docker Compose in production at `https://docs.docker.com/compose/production/`.

For more advanced production environments, you will need to dynamically distribute containers across a varying number of machines. For that, instead of Docker Compose, you will need an orchestrator like Docker Swarm mode or Kubernetes. You can find information about Docker Swarm mode at `https://docs.docker.com/engine/swarm/`, and about Kubernetes at `https://kubernetes.io/docs/home/`.

Creating a custom middleware

You already know the `MIDDLEWARE` setting, which contains the middleware for your project. You can think of it as a low-level plugin system, allowing you to implement hooks that get executed in the request/response process. Each middleware is responsible for some specific action that will be executed for all HTTP requests or responses.

Avoid adding expensive processing to middleware, since they are executed in every single request.

When an HTTP request is received, middleware is executed in order of appearance in the `MIDDLEWARE` setting. When an HTTP response has been generated by Django, the response passes through all middleware back in reverse order.

A middleware can be written as a function, as follows:

```
def my_middleware(get_response):
    def middleware(request):
        # Code executed for each request before
        # the view (and later middleware) are called.
        response = get_response(request)
        # Code executed for each request/response after
        # the view is called.
        return response
    return middleware
```

A middleware factory is a callable that takes a `get_response` callable and returns a middleware. A middleware is a callable that takes a request and returns a response, just like a view. The `get_response` callable might be the next middleware in the chain or the actual view in the case of the last listed middleware.

If any middleware returns a response without calling its `get_response` callable, it short-circuits the process; no further middleware gets executed (also not the view), and the response returns through the same layers that the request passed in through.

The order of middleware in the `MIDDLEWARE` setting is very important because middleware can depend on data set in the request by other middleware that has been executed previously.

When adding a new middleware to the `MIDDLEWARE` setting, make sure to place it in the right position. Middleware is executed in order of appearance in the setting during the request phase, and in reverse order for responses.

You can find more information about middleware at `https://docs.djangoproject.com/en/4.1/topics/http/middleware/`.

Creating a subdomain middleware

You are going to create a custom middleware to allow courses to be accessible through a custom subdomain. Each course detail URL, which looks like `https://educaproject.com/course/django/`, will also be accessible through the subdomain that makes use of the course slug, such as `https://django.educaproject.com/`. Users will be able to use the subdomain as a shortcut to access the course details. Any requests to subdomains will be redirected to each corresponding course detail URL.

Middleware can reside anywhere within your project. However, it's recommended to create a `middleware.py` file in your application directory.

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

```
from django.urls import reverse
from django.shortcuts import get_object_or_404, redirect
from .models import Course


def subdomain_course_middleware(get_response):
    """
    Subdomains for courses
    """
    def middleware(request):
        host_parts = request.get_host().split('.')
        if len(host_parts) > 2 and host_parts[0] != 'www':
            # get course for the given subdomain
            course = get_object_or_404(Course, slug=host_parts[0])
            course_url = reverse('course_detail',
                                 args=[course.slug])
            # redirect current request to the course_detail view
            url = '{}://{}{}'.format(request.scheme,
                                     '.'.join(host_parts[1:]),
                                     course_url)
            return redirect(url)
        response = get_response(request)
        return response
    return middleware
```

When an HTTP request is received, you perform the following tasks:

1. You get the hostname that is being used in the request and divide it into parts. For example, if the user is accessing `mycourse.educaproject.com`, you generate the list `['mycourse', 'educaproject', 'com']`.

2. You check whether the hostname includes a subdomain by checking whether the split generated more than two elements. If the hostname includes a subdomain, and this is not `www`, you try to get the course with the slug provided in the subdomain.
3. If a course is not found, you raise an HTTP `404` exception. Otherwise, you redirect the browser to the course detail URL.

Edit the `settings/base.py` file of the project and add `'courses.middleware.SubdomainCourseMiddleware'` at the bottom of the `MIDDLEWARE` list, as follows:

```
MIDDLEWARE = [
    # ...
    'courses.middleware.subdomain_course_middleware',
]
```

The middleware will now be executed in every request.

Remember that the hostnames allowed to serve your Django project are specified in the `ALLOWED_HOSTS` setting. Let's change this setting so that any possible subdomain of `educaproject.com` is allowed to serve your application.

Edit the `educa/settings/prod.py` file and modify the `ALLOWED_HOSTS` setting, as follows:

```
ALLOWED_HOSTS = ['.educaproject.com']
```

A value that begins with a period is used as a subdomain wildcard; `'.educaproject.com'` will match `educaproject.com` and any subdomain for this domain, for example, `course.educaproject.com` and `django.educaproject.com`.

Serving multiple subdomains with NGINX

You need NGINX to be able to serve your site with any possible subdomain. Edit the `config/nginx/default.conf.template` file and replace the two occurrences of the following line:

```
server_name  www.educaproject.com educaproject.com;
```

with the following one:

```
server_name  *.educaproject.com educaproject.com;
```

By using the asterisk, this rule applies to all subdomains of `educaproject.com`. In order to test your middleware locally, you need to add any subdomains you want to test to `/etc/hosts`. For testing the middleware with a `Course` object with the slug `django`, add the following line to your `/etc/hosts` file:

```
127.0.0.1  django.educaproject.com
```

Stop the Docker application from the shell by pressing the keys *Ctrl + C* or using the stop button in the Docker Desktop app. Then start Compose again with the command:

```
docker compose up
```

Then, open `https://django.educaproject.com/` in your browser. The middleware will find the course by the subdomain and redirect your browser to `https://educaproject.com/course/django/`.

Implementing custom management commands

Django allows your applications to register custom management commands for the `manage.py` utility. For example, you used the management commands `makemessages` and `compilemessages` in *Chapter 11*, *Adding Internationalization to Your Shop*, to create and compile translation files.

A management command consists of a Python module containing a `Command` class that inherits from `django.core.management.base.BaseCommand` or one of its subclasses. You can create simple commands or make them take positional and optional arguments as input.

Django looks for management commands in the `management/commands/` directory for each active application in the `INSTALLED_APPS` setting. Each module found is registered as a management command named after it.

You can learn more about custom management commands at `https://docs.djangoproject.com/en/4.1/howto/custom-management-commands/`.

You are going to create a custom management command to remind students to enroll on at least one course. The command will send an email reminder to users who have been registered for longer than a specified period and who aren't enrolled on any course yet.

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

```
management/
    __init__.py
    commands/
        __init__.py
        enroll_reminder.py
```

Edit the `enroll_reminder.py` file and add the following code to it:

```
import datetime
from django.conf import settings
from django.core.management.base import BaseCommand
from django.core.mail import send_mass_mail
from django.contrib.auth.models import User
from django.db.models import Count
from django.utils import timezone

class Command(BaseCommand):
    help = 'Sends an e-mail reminder to users registered more \
           than N days that are not enrolled into any courses yet'
```

```
    def add_arguments(self, parser):
        parser.add_argument('--days', dest='days', type=int)

    def handle(self, *args, **options):
        emails = []
        subject = 'Enroll in a course'
        date_joined = timezone.now().today() - \
                      datetime.timedelta(days=options['days'] or 0)
        users = User.objects.annotate(course_count=Count('courses_joined'))\
                            .filter(course_count=0,
                                    date_joined__date__lte=date_joined)
        for user in users:
            message = """Dear {},
            We noticed that you didn't enroll in any courses yet.
            What are you waiting for?""".format(user.first_name)
            emails.append((subject,
                           message,
                           settings.DEFAULT_FROM_EMAIL,
                           [user.email]))
        send_mass_mail(emails)
        self.stdout.write('Sent {} reminders'.format(len(emails)))
```

This is your `enroll_reminder` command. The preceding code is as follows:

- The `Command` class inherits from `BaseCommand`.
- You include a `help` attribute. This attribute provides a short description of the command that is printed if you run the command `python manage.py help enroll_reminder`.
- You use the `add_arguments()` method to add the `--days` named argument. This argument is used to specify the minimum number of days a user has to be registered, without having enrolled on any course, in order to receive the reminder.
- The `handle()` command contains the actual command. You get the `days` attribute parsed from the command line. If this is not set, you use `0`, so that a reminder is sent to all users that haven't enrolled on a course, regardless of when they registered. You use the `timezone` utility provided by Django to retrieve the current timezone-aware date with `timezone.now().date()`. (You can set the timezone for your project with the `TIME_ZONE` setting.) You retrieve the users who have been registered for more than the specified days and are not enrolled on any courses yet. You achieve this by annotating the QuerySet with the total number of courses each user is enrolled on. You generate the reminder email for each user and append it to the `emails` list. Finally, you send the emails using the `send_mass_mail()` function, which is optimized to open a single SMTP connection for sending all emails, instead of opening one connection per email sent.

You have created your first management command. Open the shell and run your command:

```
docker compose exec web python /code/educa/manage.py \
  enroll_reminder --days=20 --settings=educa.settings.prod
```

If you don't have a local SMTP server running, you can look at *Chapter 2*, *Enhancing Your Blog with Advanced Features*, where you configured SMTP settings for your first Django project. Alternatively, you can add the following setting to the `settings.py` file to make Django output emails to the standard output during development:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Django also includes a utility to call management commands using Python. You can run management commands from your code as follows:

```
from django.core import management
management.call_command('enroll_reminder', days=20)
```

Congratulations! You can now create custom management commands for your applications.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter17`
- Docker Compose overview – `https://docs.docker.com/compose/`
- Installing Docker Desktop – `https://docs.docker.com/compose/install/compose-desktop/`
- Official Python Docker image – `https://hub.docker.com/_/python`
- Dockerfile reference – `https://docs.docker.com/engine/reference/builder/`
- `requirements.txt` file for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/requirements.txt`
- YAML file example – `https://yaml.org/`
- `Dockerfile build` section – `https://docs.docker.com/compose/compose-file/build/`
- Docker restart policy – `https://docs.docker.com/config/containers/start-containers-automatically/`
- Docker volumes – `https://docs.docker.com/storage/volumes/`
- Docker Compose specification – `https://docs.docker.com/compose/compose-file/`
- Official PostgreSQL Docker image – `https://hub.docker.com/_/postgres`
- `wait-for-it.sh` Bash script for Docker – `https://github.com/vishnubob/wait-for-it/blob/master/wait-for-it.sh`
- Service startup order in Compose –`https://docs.docker.com/compose/startup-order/`
- Official Redis Docker image – `https://hub.docker.com/_/redis`
- WSGI documentation – `https://wsgi.readthedocs.io/en/latest/`

- List of uWSGI options – `https://uwsgi-docs.readthedocs.io/en/latest/Options.html`
- Official NGINX Docker image – `https://hub.docker.com/_/nginx`
- NGINX documentation – `https://nginx.org/en/docs/`
- `ALLOWED_HOSTS` setting – `https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts`
- Django's system check framework – `https://docs.djangoproject.com/en/4.1/topics/checks/`
- HTTP Strict Transport Security policy with Django – `https://docs.djangoproject.com/en/4.1/ref/middleware/#http-strict-transport-security`
- Django deployment checklist – `https://docs.djangoproject.com/en/4.1/howto/deployment/checklist/`
- Self-generated SSL/TLS certificate directory – `https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/educa/ssl/`
- Let's Encrypt Certificate Authority – `https://letsencrypt.org/`
- Daphne source code – `https://github.com/django/daphne`
- Using Docker Compose in production – `https://docs.docker.com/compose/production/`
- Docker Swarm mode – `https://docs.docker.com/engine/swarm/`
- Kubernetes – `https://kubernetes.io/docs/home/`
- Django middleware – `https://docs.djangoproject.com/en/4.1/topics/http/middleware/`
- Creating custom management commands – `https://docs.djangoproject.com/en/4.1/howto/custom-management-commands/`

Summary

In this chapter, you created a production environment using Docker Compose. You configured NGINX, uWSGI, and Daphne to serve your application in production. You secured your environment using SSL/TLS. You also implemented a custom middleware and you learned how to create custom management commands.

You have reached the end of this book. Congratulations! You have learned the skills required to build successful web applications with Django. This book has guided you through the process of developing real-life projects and integrating Django with other technologies. Now you are ready to create your own Django project, whether it is a simple prototype or a large-scale web application.

Good luck with your next Django adventure!

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

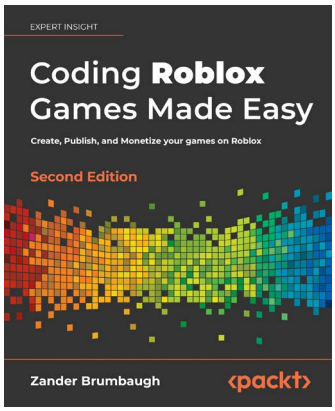
If you enjoyed this book, you may be interested in these other books by Packt:

101 UX Principles, Second Edition

Will Grant

ISBN: 9781803234885

- Work with user expectations, not against them
- Make interactive elements obvious and discoverable
- Optimize your interface for mobile
- Streamline creating and entering passwords
- Use animation with care in user interfaces
- How to handle destructive user actions



Coding Roblox Games Made Easy, Second Edition

Zander Brumbaugh

ISBN: 9781803234670

- Use Roblox Studio and other free resources
- Learn coding in Luau: basics, game systems, physics manipulation, etc
- Test, evaluate, and redesign to create bug-free and engaging games
- Use Roblox programming and rewards to make your first game
- Move from lobby to battleground, build avatars, locate weapons to fight
- Character selection, countdown timers, locate escape items, assign rewards
- Master the 3 Ms: Mechanics, Monetization, Marketing (and Metaverse)
- 50 cool things to do in Roblox

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Django 4 By Example, Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y