

13

Going Live

In the previous chapter, you created a RESTful API for your project. In this chapter, you will learn how to do the following:

- Configure a production environment
- Create a custom middleware
- Implement custom management commands

Going live to a production environment

It's time to deploy your Django project in a production environment. We are going to follow the next steps to get our project live:

1. Configure project settings for a production environment.
2. Use a PostgreSQL database.
3. Set up a web server with uWSGI and Nginx.
4. Serve static assets.
5. Secure our site with SSL.

Managing settings for multiple environments

In real world projects, you will have to deal with multiple environments. You will have at least a local and a production environment, but there might be other environments as well. Some project settings will be common to all environments, but others will have to be overridden per environment. Let's set up project settings for multiple environments, while keeping everything neatly organized.

Create a `settings/` directory inside your main `educa` project directory. Rename the `settings.py` file of your project to `settings/base.py` and create the following additional files so that the new directory looks like this:

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

These files are as follows:

- `base.py`: The base settings file that contains common and default settings
- `local.py`: Custom settings for your local environment
- `pro.py`: Custom settings for the production environment

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

```
BASE_DIR =  
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

With the following one:

```
BASE_DIR =  
os.path.dirname(os.path.dirname(os.path.abspath(os.path.join(__file__,  
os.pardir)))))
```

We have moved our settings files one directory level lower, so we need `BASE_DIR` to point to the parent directory to be correct. We achieve this by pointing to the parent directory with `os.pardir`.

Edit the `settings/local.py` file and add the following lines of code:

```
from .base import *  
  
DEBUG = True  
  
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
    }  
}
```

This is the settings file for our local environment. We import all settings defined in the `base.py` file and only define specific settings for this environment. We have copied the `DEBUG` and `DATABASES` settings from the `base.py` file, since these will be set per environment. You can remove the `DATABASES` and `DEBUG` settings from the `base.py` settings file.

Edit the `settings/pro.py` file and make it look as follows:

```
from .base import *

DEBUG = False

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

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

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

These are the settings for the production environment. Let's take a closer look at each of them:

- **DEBUG:** Setting `DEBUG` to `False` should be mandatory for any production environment. Failing to do so will result in traceback information and sensitive configuration data 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 to replace the name/e-mail tuple above with your own information.
- **ALLOWED_HOSTS:** Since `DEBUG` is `False`, Django will only allow the hosts included in this list to serve the application. This is a security measure. We have included the domain names `educaproject.com` and `www.educaproject.com`, which we will use for our site.
- **DATABASES:** We just keep this setting empty. We are going to cover database setup for production hereafter.



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.

We have placed the project settings in a different location from the default `settings.py` file. You will not be able to execute any commands with the `manage.py` tool unless you specify the settings module to use. You will need to add a `--settings` flag when you run management commands from the shell or set a `DJANGO_SETTINGS_MODULE` environment variable. Open the shell and run the following command:

```
export DJANGO_SETTINGS_MODULE=educa.settings.pro
```

This will set the `DJANGO_SETTINGS_MODULE` environment variable for the current shell session. If you want to avoid executing this command for each new shell, add this command to your shell's configuration in the `.bashrc` or `.bash_profile` file. If you don't set this variable, you will have to run management commands, including the `--settings` flag. For example:

```
python manage.py migrate --settings=educa.settings.pro
```

You have now successfully organized settings for multiple environments.

Installing PostgreSQL

Throughout this book we have been using the SQLite database. This is simple and quick to setup, but for a production environment you will need a more powerful database like PostgreSQL, MySQL, or Oracle. We are going to use PostgreSQL for our production environment. PostgreSQL is the recommended database for Django because of the features and performance it offers. Django also comes with the `django.contrib.postgres` package that allows you to take advantage of specific PostgreSQL features. You can find more information about this module at <https://docs.djangoproject.com/en/1.8/ref/contrib/postgres/>.

If you are using Linux, install the dependencies for PostgreSQL to work with Python like this:

```
sudo apt-get install libpq-dev python-dev
```

Then install PostgreSQL with the following command:

```
sudo apt-get install postgresql postgresql-contrib
```

If you are using Mac OS X or Windows, you can download PostgreSQL from <http://www.postgresql.org/download/>.

Let's create a PostgreSQL user. Open the shell and run the following commands:

```
su postgres
createuser -dP educa
```

You will be prompted for a password and permissions you want to give to this user. Enter the desired password and permissions and then create a new database with the following command:

```
createdb -E utf8 -U educa educa
```

Then edit the `settings/pro.py` file and modify the `DATABASES` setting to make it look as follows:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'educa',
        'USER': 'educa',
        'PASSWORD': '*****',
    }
}
```

Replace the preceding data with the database name and credentials for the user you created. The new database is empty. Run the following command to apply all database migrations:

```
python manage.py migrate
```

Finally, create a superuser with the following command:

```
python manage.py createsuperuser
```

Checking your project

Django includes the `check` management command to check your project anytime. This command inspects the apps installed in your Django project and outputs any errors or warnings. If you include the `--deploy` option, additional checks only relevant for production use will be triggered. Open the shell and run the following command to perform a check:

```
python manage.py check --deploy
```

You will see an output with no errors, but several warnings. This means the check was successful, but you should go through the warnings to see if there is anything more you can do to make your project safe for production. We are not going to go deeper into this, but just keep in mind that you should check your project before production use, to identify any relevant issues.

Serving Django through WSGI

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 for your application. WSGI is used to both run your project with the Django development server, and deploy your application with the server of your choice in a production environment.

You can learn more about WSGI at <http://wsgi.readthedocs.org/en/latest/>.

Installing uWSGI

Throughout this book, you have been using the Django development server to run projects in your local environment. However, you need a real web server to deploy 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.

Install uWSGI using the following command:

```
pip install uwsgi==2.0.11.1
```

If you are using Mac OS X, you can install uWSGI with the Homebrew package manager using the command `brew install uwsgi`. If you want to install uWSGI on Windows, you will need Cygwin <https://www.cygwin.com/>. However, it is recommended to use uWSGI in UNIX-based environments.

Configuring uWSGI

You can run uWSGI from the command line. Open the shell and run the following command from the `educa` project directory:

```
uwsgi --module=educa.wsgi:application \  
--env=DJANGO_SETTINGS_MODULE=educa.settings.pro \  
--http=127.0.0.1:80 \  
--uid=1000 \  
--virtualenv=/home/zenx/env/educa/
```

You might have to prepend `sudo` to this command if you don't have the required permissions.

With this command, we run uWSGI on our localhost with the following options:

- We use the `educa.wsgi:application` WSGI callable.
- We load the settings for the production environment.
- We use our virtual environment. Replace the path of the `virtualenv` option with your actual virtual environment directory. If you are not using a virtual environment, you can skip this option.

If you are not running the command within the project directory, include the option `--chdir=/path/to/educa/` with the path to your project.

Open `http://127.0.0.1/` in your browser. You should see the generated HTML, but no CSS or images are loaded. This makes sense, since we didn't configure uWSGI to serve static files.

uWSGI allows you to define custom configuration in an `.ini` file. This is more convenient than passing options through the command line. Create the following file structure inside the main `educa/` directory:

```
config/  
uwsgi.ini
```

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

```
[uwsgi]  
# variables  
projectname = educa  
base = /home/zenx/educa  
  
# configuration  
master = true  
virtualenv = /home/zenx/env/%(projectname)
```

```
pythonpath = %(base)
chdir = %(base)
env = DJANGO_SETTINGS_MODULE=%(projectname).settings.pro
module = educa.wsgi.application
socket = /tmp/%(projectname).sock
```

We define the following variables:

- `projectname`: The name of your Django project, which is `educa`.
- `base`: The absolute path to the `educa` project. Replace it with your absolute path.

These are custom variables that we will use in the uWSGI options. You can define any other variables you like as long as the name is different from uWSGI options. We set the following options:

- `master`: Enable master process.
- `virtualenv`: The path to your virtual environment. Replace this with the appropriate path.
- `pythonpath`: The path to add to your Python path.
- `chdir`: The path to your project directory, so that uWSGI changes to that directory before loading the application.
- `env`: Environment variables. We include the `DJANGO_SETTINGS_MODULE` variable, that points to the settings for the production environment.
- `module`: The WSGI module to use. We set this to the `application` callable contained in the `wsgi` module of our project.
- `socket`: The UNIX/TCP socket to bind the server.

The `socket` option is intended for communication with some third-party router such as Nginx, while the `http` option is for uWSGI to accept incoming HTTP requests and route them by itself. We are going to run uWSGI using a socket, since we are going to configure Nginx as our web server and communicate with uWSGI through the socket.

You can find the list of all available uWSGI options at <http://uwsgi-docs.readthedocs.org/en/latest/Options.html>.

Now you can run uWSGI with your custom configuration using this command:

```
uwsgi --ini config/uwsgi.ini
```

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.

Installing Nginx

When you are serving a website, you have to serve dynamic content, but you also need to serve static files such as CSS, JavaScript files, and images. While uWSGI is capable of serving static files, it adds an unnecessary overhead to HTTP requests. Therefore, it's recommended to set up a web server like Nginx in front of uWSGI, to serve static files.

Nginx is a web server focused on high concurrency, performance, and low memory usage. Nginx also acts as a reverse proxy, receiving HTTP requests and routing them to different backends. Generally, you will use a web server such as Nginx in front, for serving static files efficiently and quickly, and you will forward dynamic requests to uWSGI workers. By using Nginx, you can also apply rules and benefit from its reverse proxy capabilities.

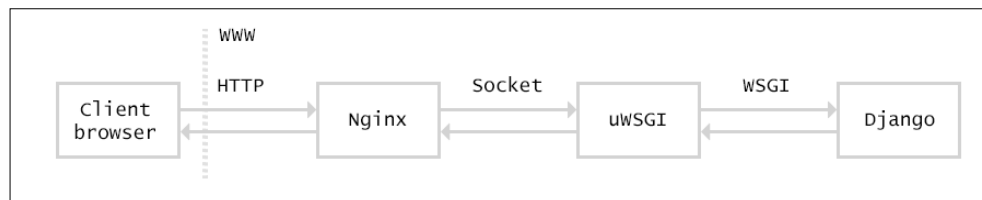
Install Nginx with the following command:

```
sudo apt-get install nginx
```

If you are using Mac OS X, you can install Nginx by using the command `brew install nginx`. You can find the Nginx binaries for Windows at <http://nginx.org/en/download.html>.

The production environment

The following schema shows how our final production environment will look:



When the client browser launches an HTTP request, the following happens:

1. Nginx receives the HTTP request.
2. If a static file is requested, Nginx serves the static file directly. If a dynamic page is requested, Nginx delegates the request to uWSGI through a socket.
3. uWSGI passes the request to Django for processing. The resulting HTTP response is passed back to Nginx, which in turn passes it back to the client browser.

Configuring Nginx

Create a new file inside the `config/` directory and name it `nginx.conf`. Add the following code to it:

```
# the upstream component nginx needs to connect to
upstream educa {
    server unix:///tmp/educa.sock;
}

server {
    listen      80;
    server_name www.educaproject.com educaproject.com;

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

This is the basic configuration for Nginx. We set up an upstream named `educa`, which points to the socket created by uWSGI. We use the `server` directive and add the following configuration:

1. We tell Nginx to listen on port 80.
2. We set the server name to both `www.educaproject.com` and `educaproject.com`. Nginx will serve incoming requests for both domains.
3. Finally, we specify that everything under the `/` path has to be routed to the `educa` socket (uWSGI). We also include the default uWSGI configuration params that come with Nginx.

You can find the Nginx documentation at <http://nginx.org/en/docs/>. The main Nginx configuration file is located at `/etc/nginx/nginx.conf`. It includes any configuration files found under `/etc/nginx/sites-enabled/`. To make Nginx load your custom configuration file, create a symbolic link as follows:

```
sudo ln -s /home/zenx/educa/config/nginx.conf /etc/nginx/sites-enabled/
educa.conf
```

Replace `/home/zenx/educa/` with your project's absolute path. Then open a shell and run uWSGI if you are not running it yet:

```
uwsgi --ini config/uwsgi.in
```

Open a second shell and run Nginx with the command:

```
service nginx start
```

Since we are using a sample domain name, we need to redirect it to our local host. Edit your `/etc/hosts` file and add the following lines to it:

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

By doing so, we are routing both hostnames to our local server. In a production server, you won't need to do this since 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 being loaded. Our production environment is almost ready.

Serving static and media assets

For the best performance, we are going to serve static assets directly with Nginx.

Edit the `settings/base.py` file and add the following code to it:

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

We need to export static assets with Django. The `collectstatic` command copies static files from all applications and stores them in the `STATIC_ROOT` directory.

Open the shell and run the following command:

```
python manage.py collectstatic
```

You will see the following output:

```
You have requested to collect static files at the destination
location as specified in your settings:
```

```
/educa/static
```

```
This will overwrite existing files!
Are you sure you want to do this?
```

Enter `yes` so that Django copies all files. You will see the following output:

```
78 static files copied to /educa/static
```

Now edit the `config/nginx.conf` file and add the following code inside the `server` directive:

```
location /static/ {
    alias /home/zenx/educa/static/;
}
location /media/ {
    alias /home/zenx/educa/media/;
}
```

Remember to replace the `/home/zenx/educa/` path with the absolute path to your project directory. These directives tell Nginx to serve the static assets located under `/static/` or `/media/` paths directly.

Reload Nginx's configuration with the following command:

```
service nginx reload
```

Open `http://educaproject.com/` in your browser. You should be able to see static files now. We have successfully configured Nginx to serve static files.

Securing connections with SSL

The SSL protocol, **Secure Sockets Layer**, is becoming the norm for serving websites through a secure connection. It's strongly encouraged that you serve your websites under HTTPS. We are going to configure an SSL certificate in Nginx to securely serve our site.

Creating a SSL certificate

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

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ssl/educa.key -out ssl/educa.crt
```

We are generating a private key and a 2048-bit SSL certificate that is valid for one year. You will be asked to enter the following information:

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []: Madrid
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Zenx IT
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []: educaproject.com
Email Address []: email@domain.com
```

You can fill in the requested data with your own information. The most important field is the **Common Name**. You have to specify the domain name for the certificate. We will use `educaproject.com`.

This will generate, inside the `ssl/` directory, an `educa.key` private key file and an `educa.crt` file, which is the actual certificate.

Configuring Nginx to use SSL

Edit the `nginx.conf` file and modify the server directive to include the following SSL directives:

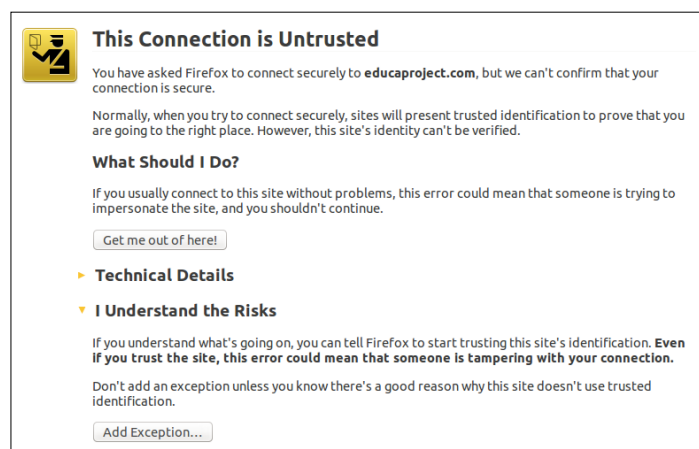
```
server {
    listen      80;
    listen      443 ssl;
    ssl_certificate /home/zenx/educa/ssl/educa.crt;
    ssl_certificate_key /home/zenx/educa/ssl/educa.key;
    server_name  www.educaproject.com educaproject.com;
    # ...
}
```

Our server now listens both to HTTP through port 80 and HTTPS through port 443. We indicate the path to the SSL certificate with `ssl_certificate` and the certificate key with `ssl_certificate_key`.

Restart Nginx with the following command:

```
sudo service nginx restart
```

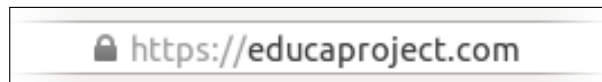
Nginx will load the new configuration. Open `https://educaproject.com/` with your browser. You should see a warning message similar to the following one:



The message might be different depending on your browser. It alerts you that your site is not using a trusted certificate; the browser cannot verify the identity of your site. This is because we signed our own certificate instead of obtaining one from a trusted Certification Authority. When you own a real domain you can apply for a trusted CA to issue an SSL 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 collaborative project which aims to simplify obtaining and renewing trusted SSL certificates for free. You can find more information at <https://letsencrypt.org/>.

Click the **Add Exception** button to let your browser know that you trust this certificate. You will see that the browser displays a lock icon next to the URL, as shown in the following screenshot:



If you click the lock icon, the SSL certificate details will be displayed.

Configuring our project for SSL

Django includes some settings specific to SSL. Edit the `settings/pro.py` settings file and add the following code to it:

```
SECURE_SSL_REDIRECT = True
CSRF_COOKIE_SECURE = True
```


These settings are as follows:

- `SECURE_SSL_REDIRECT`: Whether HTTP requests have to be redirected to HTTPS ones
- `CSRF_COOKIE_SECURE`: This has to be set to establish a secure cookie for the cross-site request forgery protection

Great! You have configured a production environment that will offer great performance to serve your project.

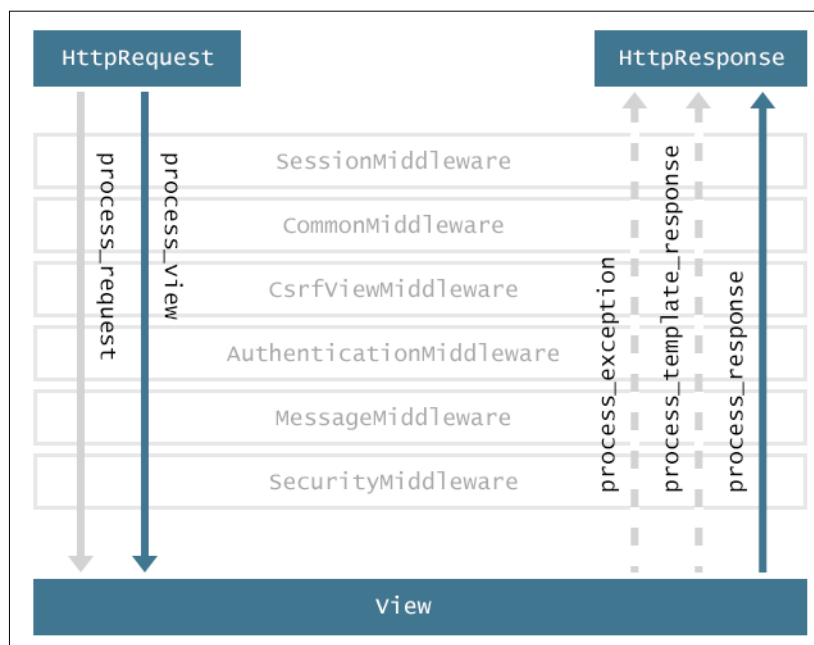
Creating custom middleware

You already know about the `MIDDLEWARE_CLASSES` setting, which contains the middleware for your project. A middleware is a class with some specific methods that are globally executed. You can think of it as a low-level plug-in system, allowing you to implement hooks that get executed in the request or response process. Each middleware is responsible for some specific action that will be executed for all requests or responses.

 Avoid adding expensive processing to middlewares, since they are executed for every single request.

When an HTTP request is received, middleware is executed in order of appearance in the `MIDDLEWARE_CLASSES` setting. When an HTTP response has been generated by Django, the middleware methods are executed in reverse order.

The following schema shows the order in which the middleware methods are executed during the request and response phases. It also displays the middleware methods that are (or might be) called:



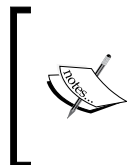
During the request phase, the following methods of the middleware are executed:

1. `process_request(request)`: Called on each request before Django decides which view to execute. `request` is an `HttpRequest` instance.
2. `process_view(request, view_func, view_args, view_kwargs)`: Called just before Django calls the view. It has access to the view function and the arguments it receives.

During the response phase, the following the middleware methods are executed:

1. `process_exception(request, exception)`: Called only when an `Exception` exception is raised by the view function.
2. `process_template_response(request, response)`: Called just after the view finishes executing, only if the response object has a `render()` method (i.e., it is a `TemplateResponse` or equivalent object).
3. `process_response(request, response)`: Called on all responses before they are returned to the browser.

The order of middleware in the `MIDDLEWARE_CLASSES` setting is very important because a middleware can depend on data set in the request by the other middleware methods that have been executed previously. Note that the `process_response()` method of a middleware is called even if `process_request()` or `process_view()` were skipped because of a previous middleware returning an HTTP response. This means that `process_response()` cannot rely on data set in the request phase. If an exception is processed by a middleware and a response is returned, the preceding middleware classes will not be executed.



When adding new middleware to the `MIDDLEWARE_CLASSES` setting, make sure to place it in the right position. The middleware methods are executed in order of appearance in the setting during the request phase, and in reverse order for the responses.

You can find more information about middleware at <https://docs.djangoproject.com/en/1.8/topics/http/middleware/>.

We are going to create custom middleware to allow courses to be accessible through a custom subdomain. Each course detail view URL, which looks like `https://educaproject.com/course/django/`, will be also accessible through the subdomain that makes use of the course slug, such as `https://django.educaproject.com/`.

Creating a subdomain middleware

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.core.urlresolvers import reverse
from django.shortcuts import get_object_or_404, redirect
from .models import Course

class SubdomainCourseMiddleware(object):
    """
    Provides subdomains for courses
    """
    def process_request(self, 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 = '{scheme}://{host}{url}'.format(request.scheme,
                                                  '.'.join(host_parts[1:]),
                                                  course_url)

            return redirect(url)
```

We create a custom middleware that implements `process_request()`. When a HTTP request is received, we perform the following tasks:

1. We 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`, we generate the list `['mycourse', 'educaproject', 'com']`.
2. We verify that the hostname includes a subdomain by checking if the split generated more than two elements. If the hostname includes a subdomain, and this is not `'www'`, we try to get the course with the `slug` provided in the subdomain.
3. If a course is not found, we raise an `Http404` exception. Otherwise, we redirect the browser to the course detail URL using the main domain.

Edit the `settings.py` file of the project and add `'courses.middleware.SubdomainCourseMiddleware'` at the bottom of the `MIDDLEWARE_CLASSES` list, as follows:

```
MIDDLEWARE_CLASSES = (  
    # ...  
    'django.middleware.security.SecurityMiddleware',  
    'courses.middleware.SubdomainCourseMiddleware',  
)
```

Our middleware will now be executed for every request.

Serving multiple subdomains with Nginx

We need Nginx to be able to serve our site with any possible subdomain. Edit the `config/nginx.conf` file by finding this line:

```
server_name www.educaproject.com educaproject.com;
```

And replacing it 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 our middleware locally, we need to add any subdomains we want to test to `/etc/hosts`. To test 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
```

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, we used the management commands `makemessages` and `compilemessages` in *Chapter 9, Extending 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.BaseCommand`. 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/1.8/howto/custom-management-commands/>.

We are going to create a custom management command to remind students to enroll at least in one course. The command will send an e-mail reminder to users that have been registered longer than a specified period, but are not enrolled in 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

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 = datetime.date.today() - datetime.
            timedelta(days=options['days'])
```

```
users = User.objects.annotate(course_count=Count('courses_
enrolled')).filter(course_count=0, date_joined__lte=date_joined)
for user in users:
    message = 'Dear {},\n\nWe 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' % len(emails))
```

This is our `enroll_reminder` command. The preceding code is as follows:

- The `Command` class inherits from `BaseCommand`.
- We include a `help` attribute. This attribute provides a short description for the command that is printed if you run the command `python manage.py help enroll_reminder`.
- We 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 in any course in order to receive the reminder.
- The `handle()` method contains the actual command. We get the `days` attribute parsed from the command line. We retrieve the users that have been registered for more than the specified number of days, but who are not enrolled into any courses yet. We achieve this by annotating the `QuerySet` with the total number of courses a user is enrolled in. We generate the reminder email for each user and append it to the `emails` list. Finally, we send the e-mails using the `send_mass_mail()` function, which is optimized to open a single SMTP connection to send all e-mails, instead of opening one connection per e-mail sent.

You have created your first management command. Open the shell and run your command:

```
python manage.py enroll_reminder --days=20
```

If you don't have a local SMTP server running, you can take a look at *Chapter 2, Enhancing Your Blog with Advanced Features*, where we configured the SMTP settings for our first Django project. Alternatively, you can add the the following line to the `settings.py` file to make Django output e-mails to the standard output during development:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Let's schedule our management command, so that the server runs it every day at 8 a.m. If you are using an UNIX based system such as Linux or Mac OS X, open the shell and run `crontab -e` to edit your cron. Add the following line to it:

```
0 8 * * * python /path/to/educa/manage.py enroll_reminder --days=20
--settings=educa.settings.pro
```

If you are not familiar with **Cron**, you can find information about it at <https://en.wikipedia.org/wiki/Cron>.

If you are using Windows, you can schedule tasks using the Task scheduler. You can find more information about it at <http://windows.microsoft.com/en-au/windows/schedule-task#1TC=windows-7>.

Another option for executing actions periodically is to create tasks and schedule them with Celery. Remember that we used Celery in *Chapter 7, Building an Online Shop*, to execute asynchronous tasks. Instead of creating management commands and scheduling them with Cron, you can create asynchronous tasks and execute them with the Celery beat scheduler. You can learn more about scheduling periodic tasks with Celery at <http://celery.readthedocs.org/en/latest/userguide/periodic-tasks.html>.



Use management commands for standalone scripts that you want to schedule with Cron or the Windows scheduled tasks control panel.

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 and schedule them if needed.

Summary

In this chapter, you configured a production environment using uWSGI and Nginx. You also implemented a custom middleware and you have learned how to create custom management commands.

