Chapter 16: django.contrib

One of the many strengths of Python is its "batteries included" philosophy: when you install Python, it comes with a large standard library of packages that you can start using immediately, without having to download anything else. Django aims to follow this philosophy, and it includes its own standard library of add-ons useful for common Web development tasks. This chapter covers that collection of add-ons.

The Django Standard Library

Django's standard library lives in the package `django.contrib`. Within each subpackage is a separate piece of add-on functionality. These pieces are not necessarily related, but some `django.contrib` subpackages may require other ones.

There's no hard requirement for the types of functionality in `django.contrib`. Some of the packages include models (and hence require you to install their database tables into your database), but others consist solely of middleware or template tags.

The single characteristic the `django.contrib` packages have in common is this: if you were to remove the`django.contrib` package entirely, you could still use Django's fundamental features with no problems. When the Django developers add new functionality to the framework, they use this rule of thumb in deciding whether the new functionality should live in `django.contrib` or elsewhere.

`django.contrib` consists of these packages:

- `admin`: The Django admin site. See Chapter 6.
- `admindocs`: Auto-documentation for the Django admin site. This book doesn't cover this feature; check the official Django documentation.
- `auth`: Django's authentication framework. See Chapter 14.
- `comments`: A comments application. This book doesn't cover this feature; check the official Django documentation.
- `contenttypes`: A framework for hooking into "types" of content, where each installed Django model is a separate content type. This framework is used internally by other "contrib" applications and is mostly intended for very advanced Django developers. Those developers should find out more about this application by reading the source code in `django/contrib/contenttypes`.
- `csrf`: Protection against Cross-Site Request Forgery (CSRF). See the later section titled "CSRF Protection."
- `databrowse`: A Django application that lets you browse your data. This book doesn't cover this feature; check the official Django documentation.
- `flatpages`: A framework for managing simple "flat" HTML content in a database. See the later section titled "Flatpages."
- `formtools`: A number of useful higher-level libraries for dealing with common patterns in forms. This book doesn't cover this feature; check the official Django documentation.
- `gis`: Extensions to Django that provide for GIS (Geographic Information Systems) support. These, for example, allow your Django models to store geographic data and perform geographic queries.

This is a large, complex library and isn't covered in this book. See http://geodjango.org/ for documentation.

- `humanize`: A set of Django template filters useful for adding a "human touch" to data. See the later section titled "Humanizing Data."
- `localflavor`: Assorted pieces of code that are useful for particular countries or cultures. For example, this includes ways to validate U.S. ZIP codes or Icelandic identification numbers.
- `markup`: A set of Django template filters that implement a number of common markup languages. See the later section titled "Markup Filters."
- `redirects`: A framework for managing redirects. See the later section titled "Redirects."
- `sessions`: Django's session framework. See Chapter 14.
- `sitemaps`: A framework for generating sitemap XML files. See Chapter 13.
- `sites`: A framework that lets you operate multiple Web sites from the same database and Django installation. See the next section, "Sites."
- `syndication`: A framework for generating syndication feeds in RSS and Atom. See Chapter 13.
- `webdesign`: Django add-ons that are particularly useful to Web *designers* (as opposed to developers). As of this writing, this included only a single template tag, `{% lorem %}`. Check the Django documentation for information.

The rest of this chapter goes into detail a number of `django.contrib` packages that we haven't yet covered in this book.

Sites

Django's sites system is a generic framework that lets you operate multiple Web sites from the same database and Django project. This is an abstract concept, and it can be tricky to understand, so we'll start with a couple of scenarios where it would be useful.

Scenario 1: Reusing Data on Multiple Sites

As we explained in Chapter 1, the Django-powered sites LJWorld.com and Lawrence.com are operated by the same news organization: the *Lawrence Journal-World* newspaper in Lawrence, Kansas. LJWorld.com focuses on news, while Lawrence.com focuses on local entertainment. But sometimes editors want to publish an article on *both* sites.

The brain-dead way of solving the problem would be to use a separate database for each site and to require site producers to publish the same story twice: once for LJWorld.com and again for Lawrence.com. But that's inefficient for site producers, and it's redundant to store multiple copies of the same story in the database.

The better solution? Both sites use the same article database, and an article is associated with one or more sites via a many-to-many relationship. The Django sites framework provides the database table to which articles can be related. It's a hook for associating data with one or more "sites."

Scenario 2: Storing Your Site Name/Domain in One Place

LJWorld.com and Lawrence.com both have e-mail alert functionality, which lets readers sign up to get notifications when news happens. It's pretty basic: a reader signs up on a Web form, and he immediately gets an e-mail saying, "Thanks for your subscription."

It would be inefficient and redundant to implement this signup-processing code twice, so the sites use the same code behind the scenes. But the "Thank you for your subscription" notice needs to be different for each site. By using `Site` objects, we can abstract the thank-you notice to use the values of the current site's `name` (e.g., `'LJWorld.com'`) and `domain` (e.g., `'www.ljworld.com'`).

The Django sites framework provides a place for you to store the `name` and `domain` for each site in your Django project, which means you can reuse those values in a generic way.

How to Use the Sites Framework

The sites framework is more a series of conventions than a framework. The whole thing is based on two simple concepts:

- The `Site` model, found in `django.contrib.sites`, has `domain` and `name` fields.
- The `SITE_ID` setting specifies the database ID of the `Site` object associated with that particular settings file.

How you use these two concepts is up to you, but Django uses them in a couple of ways automatically via simple conventions.

To install the sites application, follow these steps:

1. Add `'django.contrib.sites'` to your `INSTALLED_APPS`.
2. Run the command `manage.py syncdb` to install the `django_site` table into your database. This will also create a default site object, with the domain `example.com`.
3. Change the `example.com` site to your own domain, and add any other `Site` objects, either through the Django admin site or via the Python API. Create a `Site` object for each site/domain that this Django project powers.
4. Define the `SITE_ID` setting in each of your settings files. This value should be the database ID of the`Site` object for the site powered by that settings file.

The Sites Framework's Capabilities

The sections that follow describe the various things you can do with the sites framework.

Reusing Data on Multiple Sites

To reuse data on multiple sites, as explained in the first scenario, just create a `ManyToManyField` to `Site` in your models, for example:

```
from django.db import models

from django.contrib.sites.models import Site


class Article(models.Model):

    headline = models.CharField(max_length=200)
```

```
    # ...

    sites = models.ManyToManyField(Site)
```

That's the infrastructure you need to associate articles with multiple sites in your database. With that in place, you can reuse the same Django view code for multiple sites. Continuing the `Article` model example, here's what an `article_detail` view might look like:

```
from django.conf import settings

from django.shortcuts import get_object_or_404

from mysite.articles.models import Article


def article_detail(request, article_id):

    a = get_object_or_404(Article, id=article_id,
sites__id=settings.SITE_ID)

    # ...
```

This view function is reusable because it checks the article's site dynamically, according to the value of the`SITE_ID` setting.

For example, say LJWorld.com's settings file has a `SITE_ID` set to `1`, and Lawrence.com's settings file has a`SITE_ID` set to `2`. If this view is called when LJWorld.com's settings file is active, then it will limit the article lookup to articles in which the list of sites includes LJWorld.com.

Associating Content with a Single Site

Similarly, you can associate a model to the `Site` model in a many-to-one relationship using `ForeignKey`.

For example, if each article is associated with only a single site, you could use a model like this:

```
from django.db import models

from django.contrib.sites.models import Site


class Article(models.Model):

    headline = models.CharField(max_length=200)

    # ...
```

```
    site = models.ForeignKey(Site)
```

This has the same benefits as described in the last section.

Hooking Into the Current Site from Views

On a lower level, you can use the sites framework in your Django views to do particular things based on the site in which the view is being called, for example:

```
from django.conf import settings


def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.
```

Of course, it's ugly to hard-code the site IDs like that. A slightly cleaner way of accomplishing the same thing is to check the current site's domain:

```
from django.conf import settings
from django.contrib.sites.models import Site


def my_view(request):
    current_site =
Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

The idiom of retrieving the `Site` object for the value of `settings.SITE_ID` is quite common, so the `Site`model's manager (`Site.objects`) has a `get_current()` method. This example is equivalent to the previous one:

```
from django.contrib.sites.models import Site


def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

Note

In this final example, you don't have to import `django.conf.settings`.

Getting the Current Domain for Display

For a DRY (Don't Repeat Yourself) approach to storing your site's name and domain name, as explained in "Scenario 2: Storing Your Site Name/Domain in One Place," just reference the `name` and `domain` of the current `Site` object. For example:

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail


def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' %
current_site.name,
        'Thanks for your subscription. We appreciate
it.\n\n-The %s team.' % current_site.name,
        'editor@%s' % current_site.domain,
        [user_email])
```

```
    # ...
```

Continuing our ongoing example of LJWorld.com and Lawrence.com, on Lawrence.com this e-mail has the subject line "Thanks for subscribing to lawrence.com alerts." On LJWorld.com, the e-mail has the subject line "Thanks for subscribing to LJWorld.com alerts." This same site-specific behavior is applied to the e-mails' message body.

An even more flexible (but more heavyweight) way of doing this would be to use Django's template system. Assuming Lawrence.com and LJWorld.com have different template directories (`TEMPLATE_DIRS`), you could simply delegate to the template system like so:

```
from django.core.mail import send_mail
from django.template import loader, Context


def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    subject =
loader.get_template('alerts/subject.txt').render(Contex
t({}))
    message =
loader.get_template('alerts/message.txt').render(Contex
t({}))
    send_mail(subject, message, 'do-not-
reply@example.com', [user_email])
    # ...
```

In this case, you have to create `subject.txt` and `message.txt` templates in both the LJWorld.com and Lawrence.com template directories. As mentioned previously, that gives you more flexibility, but it's also more complex.

It's a good idea to exploit the `Site` objects as much as possible to remove unneeded complexity and redundancy.

CurrentSiteManager

If `Site` objects play a key role in your application, consider using the `CurrentSiteManager` in your model(s). It's a model manager (see Chapter 10) that automatically filters its queries to include only objects associated with the current `Site`.

Use `CurrentSiteManager` by adding it to your model explicitly. For example:

```
from django.db import models

from django.contrib.sites.models import Site

from django.contrib.sites.managers import
CurrentSiteManager


class Photo(models.Model):

    photo = models.FileField(upload_to='/home/photos')

    photographer_name =
models.CharField(max_length=100)

    pub_date = models.DateField()

    site = models.ForeignKey(Site)

    objects = models.Manager()

    on_site = CurrentSiteManager()
```

With this model, `Photo.objects.all()` will return all `Photo` objects in the database, but`Photo.on_site.all()` will return only the `Photo` objects associated with the current site, according to the`SITE_ID` setting.

In other words, these two statements are equivalent:

```
Photo.objects.filter(site=settings.SITE_ID)

Photo.on_site.all()
```

How did `CurrentSiteManager` know which field of `Photo` was the `Site`? It defaults to looking for a field called`site`. If your model has a `ForeignKey` or `ManyToManyField` called something *other* than `site`, you need to explicitly pass that as the parameter to `CurrentSiteManager`. The following model, which has a field called`publish_on`, demonstrates this:

```
from django.db import models

from django.contrib.sites.models import Site

from django.contrib.sites.managers import
CurrentSiteManager


```

```
class Photo(models.Model):

    photo = models.FileField(upload_to='/home/photos')

    photographer_name =
models.CharField(max_length=100)

    pub_date = models.DateField()

    publish_on = models.ForeignKey(Site)

    objects = models.Manager()

    on_site = CurrentSiteManager('publish_on')
```

If you attempt to use `CurrentSiteManager` and pass a field name that doesn't exist, Django will raise a`ValueError`.

Note

You'll probably want to keep a normal (non-site-specific) `Manager` on your model, even if you use `CurrentSiteManager`. As explained in Appendix B, if you define a manager manually, then Django won't create the automatic `objects = models.Manager()` manager for you.

Also, certain parts of Django – namely, the Django admin site and generic views – use whichever manager is defined *first* in the model, so if you want your admin site to have access to all objects (not just site-specific ones), put `objects = models.Manager()` in your model, before you define `CurrentSiteManager`.

How Django Uses the Sites Framework

Although it's not required that you use the sites framework, it's encouraged, because Django takes advantage of it in a few places. Even if your Django installation is powering only a single site, you should take a few seconds to create the site object with your `domain` and `name`, and point to its ID in your `SITE_ID`setting.

Here's how Django uses the sites framework:

- In the redirects framework (see the later section "Redirects"), each redirect object is associated with a particular site. When Django searches for a redirect, it takes into account the current `SITE_ID`.
- In the comments framework, each comment is associated with a particular site. When a comment is posted, its `site` is set to the current `SITE_ID`, and when comments are listed via the appropriate template tag, only the comments for the current site are displayed.
- In the flatpages framework (see the later section "Flatpages"), each flatpage is associated with a particular site. When a flatpage is created, you specify its `site`, and the flatpage middleware checks the current `SITE_ID` in retrieving flatpages to display.
- In the syndication framework (see Chapter 13), the templates for `title` and `description` automatically have access to a variable `{{ site }}`, which is

the `Site` object representing the current site. Also, the hook for providing item URLs will use the `domain` from the current `Site` object if you don't specify a fully qualified domain.

- In the authentication framework (see Chapter 14), the `django.contrib.auth.views.login` view passes the current `Site` name to the template as `{{ site_name }}` and the current `Site` object as `{{ site }}`.

Flatpages

Often you'll have a database-driven Web application up and running, but you'll need to add a couple of one-off static pages, such as an About page or a Privacy Policy page. It would be possible to use a standard Web server such as Apache to serve these files as flat HTML files, but that introduces an extra level of complexity into your application, because then you have to worry about configuring Apache, you have to set up access for your team to edit those files, and you can't take advantage of Django's template system to style the pages.

The solution to this problem is Django's flatpages application, which lives in the package`django.contrib.flatpages`. This application lets you manage such one-off pages via Django's admin site, and it lets you specify templates for them using Django's template system. It uses Django models behind the scenes, which means it stores the pages in a database, just like the rest of your data, and you can access flatpages with the standard Django database API.

Flatpages are keyed by their URL and site. When you create a flatpage, you specify which URL it's associated with, along with which site(s) it's on. (For more on sites, see the "Sites" section.)

Using Flatpages

To install the flatpages application, follow these steps:

1. Add `'django.contrib.flatpages'` to your `INSTALLED_APPS`. `django.contrib.flatpages` depends on`django.contrib.sites`, so make sure the both packages are in `INSTALLED_APPS`.
2. Add `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` to your`MIDDLEWARE_CLASSES` setting.
3. Run the command `manage.py syncdb` to install the two required tables into your database.

The flatpages application creates two tables in your database: `django_flatpage` and`django_flatpage_sites`. `django_flatpage` simply maps a URL to a title and bunch of text content.`django_flatpage_sites` is a many-to-many table that associates a flatpage with one or more sites.

The application comes with a single `FlatPage` model, defined in `django/contrib/flatpages/models.py`. It looks something like this:

```
from django.db import models

from django.contrib.sites.models import Site


class FlatPage(models.Model):
```

```
    url = models.CharField(max_length=100,
db_index=True)

    title = models.CharField(max_length=200)

    content = models.TextField(blank=True)

    enable_comments = models.BooleanField()

    template_name = models.CharField(max_length=70,
blank=True)

    registration_required = models.BooleanField()

    sites = models.ManyToManyField(Site)
```

Let's examine these fields one at a time:

- `url`: The URL at which this flatpage lives, excluding the domain name but including the leading slash (e.g., `/about/contact/`).
- `title`: The title of the flatpage. The framework doesn't do anything special with this. It's your responsibility to display it in your template.
- `content`: The content of the flatpage (i.e., the HTML of the page). The framework doesn't do anything special with this. It's your responsibility to display it in the template.
- `enable_comments`: Whether to enable comments on this flatpage. The framework doesn't do anything special with this. You can check this value in your template and display a comment form if needed.
- `template_name`: The name of the template to use for rendering this flatpage. This is optional; if it's not given or if this template doesn't exist, the framework will fall back to the template`flatpages/default.html`.
- `registration_required`: Whether registration is required for viewing this flatpage. This integrates with Django's authentication/user framework, which is explained further in Chapter 14.
- `sites`: The sites that this flatpage lives on. This integrates with Django's sites framework, which is explained in the "Sites" section of this chapter.

You can create flatpages through either the Django admin interface or the Django database API. For more information on this, see the section "Adding, Changing, and Deleting Flatpages."

Once you've created flatpages, `FlatpageFallbackMiddleware` does all of the work. Each time any Django application raises a 404 error, this middleware checks the flatpages database for the requested URL as a last resort. Specifically, it checks for a flatpage with the given URL with a site ID that corresponds to the`SITE_ID` setting.

If it finds a match, it loads the flatpage's template or `flatpages/default.html` if the flatpage has not specified a custom template. It passes that template a single context variable, `flatpage`, which is the`FlatPage` object. It uses `RequestContext` in rendering the template.

If `FlatpageFallbackMiddleware` doesn't find a match, the request continues to be processed as usual.

Note

This middleware only gets activated for 404 (page not found) errors – not for 500 (server error) or other error responses. Also note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `FlatpageFallbackMiddleware` at or near the end of the list, because it's a last resort.

Adding, Changing, and Deleting Flatpages

You can add, change and delete flatpages in two ways:

Via the Admin Interface

If you've activated the automatic Django admin interface, you should see a "Flatpages" section on the admin index page. Edit flatpages as you would edit any other object in the system.

Via the Python API

As described previously, flatpages are represented by a standard Django model that lives in`django/contrib/flatpages/models.py`. Hence, you can access flatpage objects via the Django database API, for example:

```
>>> from django.contrib.flatpages.models import
FlatPage

>>> from django.contrib.sites.models import Site

>>> fp = FlatPage.objects.create(

...     url='/about/',

...     title='About',

...     content='<p>About this site...</p>',

...     enable_comments=False,

...     template_name='',

...     registration_required=False,

... )

>>> fp.sites.add(Site.objects.get(id=1))

>>> FlatPage.objects.get(url='/about/')

<FlatPage: /about/ -- About>
```

Using Flatpage Templates

By default, flatpages are rendered via the template `flatpages/default.html`, but you can override that for a particular flatpage with the `template_name` field on the `FlatPage` object.

Creating the `flatpages/default.html` template is your responsibility. In your template directory, just create a `flatpages` directory containing a `default.html` file.

Flatpage templates are passed a single context variable, `flatpage`, which is the flatpage object.

Here's a sample `flatpages/default.html` template:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN"

    "http://www.w3.org/TR/REC-html40/loose.dtd">

<html>

<head>

<title>{{ flatpage.title }}</title>

</head>

<body>

{{ flatpage.content|safe }}

</body>

</html>
```

Note that we've used the `safe` template filter to allow `flatpage.content` to include raw HTML and bypass auto-escaping.

Redirects

Django's redirects framework lets you manage redirects easily by storing them in a database and treating them as any other Django model object. For example, you can use the redirects framework to tell Django, "Redirect any request to `/music/` to `/sections/arts/music/`." This comes in handy when you need to move things around on your site; Web developers should do whatever is necessary to avoid broken links.

Using the Redirects Framework

To install the redirects application, follow these steps:

1. Add `'django.contrib.redirects'` to your `INSTALLED_APPS`.
2. Add `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` to your`MIDDLEWARE_CLASSES` setting.
3. Run the command `manage.py syncdb` to install the single required table into your database.

`manage.py syncdb` creates a `django_redirect` table in your database. This is a simple lookup table with`site_id`, `old_path`, and `new_path` fields.

You can create redirects through either the Django admin interface or the Django database API. For more, see the section "Adding, Changing, and Deleting Redirects."

Once you've created redirects, the `RedirectFallbackMiddleware` class does all of the work. Each time any Django application raises a 404 error, this middleware checks the redirects database for the requested URL as a last resort. Specifically, it checks for a redirect with the given `old_path` with a site ID that corresponds to the `SITE_ID` setting. (See the earlier section "Sites" for more information on `SITE_ID` and the sites framework.) Then it follows these steps:

- If it finds a match, and `new_path` is not empty, it redirects to `new_path`.
- If it finds a match, and `new_path` is empty, it sends a 410 ("Gone") HTTP header and an empty (contentless) response.
- If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404 errors – not for 500 errors or responses of any other status code.

Note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `RedirectFallbackMiddleware`toward the end of the list, because it's a last resort.

Note

If you're using both the redirect and flatpage fallback middleware, consider which one (redirect or flatpage) you'd like checked first. We suggest flatpages before redirects (thus putting the flatpage middleware before the redirect middleware), but you might feel differently.

Adding, Changing, and Deleting Redirects

You can add, change and delete redirects in two ways:

Via the Admin Interface

If you've activated the automatic Django admin interface, you should see a "Redirects" section on the admin index page. Edit redirects as you would edit any other object in the system.

Via the Python API

Redirects are represented by a standard Django model that lives in `django/contrib/redirects/models.py`. Hence, you can access redirect objects via the Django database API, for example:

```
>>> from django.contrib.redirects.models import
Redirect

>>> from django.contrib.sites.models import Site

>>> red = Redirect.objects.create(
```

```
...     site=Site.objects.get(id=1),
...     old_path='/music/',
...     new_path='/sections/arts/music/',
... )
>>> Redirect.objects.get(old_path='/music/')
<Redirect: /music/ ---> /sections/arts/music/>
```

CSRF Protection

The `django.contrib.csrf` package protects against Cross-Site Request Forgery (CSRF).

CSRF, also known as "session riding," is a Web site security exploit. It happens when a malicious Web site tricks a user into unknowingly loading a URL from a site at which that user is already authenticated, hence taking advantage of the user's authenticated status. This can be a bit tricky to understand at first, so we walk through two examples in this section.

A Simple CSRF Example

Suppose you're logged in to a webmail account at `example.com`. This webmail site has a Log Out button that points to the URL `example.com/logout` – that is, the only action you need to take in order to log out is to visit the page `example.com/logout`.

A malicious site can coerce you to visit the URL `example.com/logout` by including that URL as a hidden`<iframe>` on its own (malicious) page. Thus, if you're logged in to the `example.com` webmail account and visit the malicious page that has an `<iframe>` to `example.com/logout`, the act of visiting the malicious page will log you out from `example.com`.

Clearly, being logged out of a webmail site against your will is not a terrifying breach of security, but this same type of exploit can happen to *any* site that trusts users, such as an online banking site or an e-commerce site, where the exploit could be used to initiate an order or payment without the user's knowledge.

A More Complex CSRF Example

In the previous example, `example.com` was partially at fault because it allowed a state change (i.e., logging the user out) to be requested via the HTTP `GET` method. It's much better practice to require an HTTP `POST`for any request that changes state on the server. But even Web sites that require `POST` for state-changing actions are vulnerable to CSRF.

Suppose `example.com` has upgraded its Log Out functionality so that it's a `<form>` button that is requested via `POST` to the URL `example.com/logout`. Furthermore, the logout `<form>` includes this hidden field:

```
<input type="hidden" name="confirm" value="true">
```

This ensures that a simple `POST` to the URL `example.com/logout` won't log a user out; in order for a user to log out, the user must request `example.com/logout` via `POST` *and* send the `confirm` `POST` variable with a value of `'true'`.

Well, despite the extra security, this arrangement can still be exploited by CSRF – the malicious page just needs to do a little more work. Attackers can create an entire form targeting your site, hide it in an invisible`<iframe>`, and then use JavaScript to submit that form automatically.

Preventing CSRF

How, then, can your site protect itself from this exploit? The first step is to make sure all `GET` requests are free of side effects. That way, if a malicious site includes one of your pages as an `<iframe>`, it won't have a negative effect.

That leaves `POST` requests. The second step is to give each `POST` `<form>` a hidden field whose value is secret and is generated from the user's session ID. Then, when processing the form on the server side, check for that secret field and raise an error if it doesn't validate.

This is exactly what Django's CSRF prevention layer does, as explained in the sections that follow.

Using the CSRF Middleware

The `django.contrib.csrf` package contains only one module: `middleware.py`. This module contains a Django middleware class, `CsrfMiddleware`, which implements the CSRF protection.

To activate this CSRF protection, add `'django.contrib.csrf.middleware.CsrfMiddleware'` to the`MIDDLEWARE_CLASSES` setting in your settings file. This middleware needs to process the response *after*`SessionMiddleware`, so `CsrfMiddleware` must appear *before* `SessionMiddleware` in the list (because the response middleware is processed last-to-first). Also, it must process the response before the response gets compressed or otherwise mangled, so `CsrfMiddleware` must come after `GZipMiddleware`. Once you've added that to your `MIDDLEWARE_CLASSES` setting, you're done. See the section "Order of MIDDLEWARE_CLASSES" in Chapter 15 for more explanation.

In case you're interested, here's how `CsrfMiddleware` works. It does these two things:

1. It modifies outgoing requests by adding a hidden form field to all `POST` forms, with the name`csrfmiddlewaretoken` and a value that is a hash of the session ID plus a secret key. The middleware does *not* modify the response if there's no session ID set, so the performance penalty is negligible for requests that don't use sessions.
2. On all incoming `POST` requests that have the session cookie set, it checks that `csrfmiddlewaretoken` is present and correct. If it isn't, the user will get a 403 `HTTP` error. The content of the 403 error page is the message "Cross Site Request Forgery detected. Request aborted."

   This ensures that only forms originating from your Web site can be used to POST data back.

   This middleware deliberately targets only HTTP `POST` requests (and the corresponding POST forms). As we explained, `GET` requests ought never to have side effects; it's your own responsibility to ensure this.

`POST` requests not accompanied by a session cookie are not protected, but they don't *need* to be protected, because a malicious Web site could make these kind of requests anyway.

To avoid altering non-HTML requests, the middleware checks the response's `Content-Type` header before modifying it. Only pages that are served as `text/html` or `application/xml+xhtml` are modified.

Limitations of the CSRF Middleware

`CsrfMiddleware` requires Django's session framework to work. (See Chapter 14 for more on sessions.) If you're using a custom session or authentication framework that manually manages session cookies, this middleware will not help you.

If your application creates HTML pages and forms in some unusual way (e.g., if it sends fragments of HTML in JavaScript `document.write` statements), you might bypass the filter that adds the hidden field to the form. In this case, the form submission will always fail. (This happens because `CsrfMiddleware` uses a regular expression to add the `csrfmiddlewaretoken` field to your HTML before the page is sent to the client, and the regular expression sometimes cannot handle wacky HTML.) If you suspect this might be happening, just view the source in your Web browser to see whether `csrfmiddlewaretoken` was inserted into your `<form>`.

For more CSRF information and examples, visit http://en.wikipedia.org/wiki/CSRF

Humanizing Data

The package `django.contrib.humanize` holds a set of Django template filters useful for adding a "human touch" to data. To activate these filters, add `'django.contrib.humanize'` to your `INSTALLED_APPS`. Once you've done that, use `{% load humanize %}` in a template, and you'll have access to the filters described in the following sections.

apnumber

For numbers 1 through 9, this filter returns the number spelled out. Otherwise, it returns the numeral. This follows Associated Press style.

Examples:

- 1 becomes "one".
- 2 becomes "two".
- 10 becomes "10".

You can pass in either an integer or a string representation of an integer.

intcomma

This filter converts an integer to a string containing commas every three digits.

Examples:

- 4500 becomes "4,500".
- 45000 becomes "45,000".
- 450000 becomes "450,000".

- 4500000 becomes “4,500,000”.

You can pass in either an integer or a string representation of an integer.

intword

This filter converts a large integer to a friendly text representation. It works best for numbers over 1 million.

Examples:

- 1000000 becomes “1.0 million”.
- 1200000 becomes “1.2 million”.
- 1200000000 becomes “1.2 billion”.

Values up to 1 quadrillion (1,000,000,000,000,000) are supported.

You can pass in either an integer or a string representation of an integer.

ordinal

This filter converts an integer to its ordinal as a string.

Examples:

- 1 becomes “1st”.
- 2 becomes “2nd”.
- 3 becomes “3rd”.
- 254 becomes “254th”.

You can pass in either an integer or a string representation of an integer.

Markup Filters

The package `django.contrib.markup` includes a handful of Django template filters, each of which implements a common markup languages:

- `textile`: Implements Textile (http://en.wikipedia.org/wiki/Textile_%28markup_language%29)
- `markdown`: Implements Markdown (http://en.wikipedia.org/wiki/Markdown)
- `restructuredtext`: Implements ReStructured Text (http://en.wikipedia.org/wiki/ReStructuredText)

In each case, the filter expects formatted markup as a string and returns a string representing the marked-up text. For example, the `textile` filter converts text that is marked up in Textile format to HTML:

```
{% load markup %}

{{ object.content|textile }}
```

To activate these filters, add `'django.contrib.markup'` to your `INSTALLED_APPS` setting. Once you've done that, use `{% load markup %}` in a template, and you'll have access to these filters. For more documentation, read the source code in `django/contrib/markup/templatetags/markup.py`.