© Daniel Rubio 2017

Daniel Rubio, Beginning Django, https://doi.org/10.1007/978-1-4842-2787-9_2

2. Django Urls and Views

Daniel Rubio[1]

(1) F. Bahia, Ensenada, Baja California, Mexico

In Chapter 1 you learned about the core building blocks in Django, inclduing what are views, models, and urls. In this chapter, you'll learn more about Django urls, which are the entry point into a Django application workflow. You'll learn how to create complex url regular expressions, how to use url values in view methods and templates, how to structure and manage urls, and how to name urls.

After urls, Django views represent the next step in almost all Django workflows, where views are charged with inspecting requests, executing business logic, querying a database and validating data, as well as generating responses. In this chapter, you'll learn how to create Django views with optional parameters, the structure of view requests and responses, how to use middleware with views, and how to create class-based views.

Url Regular Expressions

Regular expressions provide a powerful approach in all programming languages to determine patterns. However, with power also comes complexity, to the point that there are entire books written on the topic of regular expressions.[1]

Although most Django urls will never exceed a fraction of the complexity illustrated in many regular expression books, it's important that you understand some of the underlying behaviors and most common patterns of regular expressions in Django urls.

PRECEDENCE RULE : GRANULAR URLS FIRST, BROAD URLS LAST

Django urls need to follow a certain order and syntax to work correctly. Broad url regular expressions should be declared last and only after more granular url regular expressions.

This is because Django url regular expression matching doesn't use short-circuiting behavior, like a nested conditional statement (e.g., if/elif/elif/elif/else) where as soon as one condition is met, the remaining options are ignored. In Django urls if there's more than one matching regular expression for an incoming url request, it will be the top-most one's action that gets triggered. Precedence for matching url regular expressions is given from top (i.e., first declared) to bottom (i.e., last declared).

You shouldn't underestimate how easy it can be to introduce two url regular expressions that match the same pattern, particularly if you've never done regular expressions since the syntax can be cryptic. Listing 2-1 illustrates the right way to declare Django urls, with more granular regular expressions toward the top and broad regular expressions toward the bottom.

Listing 2-1. **Correct precedence for Django url regular expressions**

```
from django.views.generic import TemplateVieww

urlpatterns = [
    url(r'^about/index/',TemplateView.as_view(template_name='index.html'
    url(r'^about/',TemplateView.as_view(template_name='about.html')),
]
```

Based on Listing 2-1, let's walk through what happens if Django receives a request for the url `/about/index/`. Initially Django matches the last regular expression, which says 'match `^about/`'. Next, Django continues upward inspecting the regular expressions and reaches 'match `^about/index/`' that is an exact match to the request url `/about/index/` and therefore triggers this action to send control to the `index.html` template.

Now let's walk through a request for the url `/about/`. Initially Django matches the last regular expression that says 'match `^about/`'. Next, Django continues upward inspecting the regular expressions for a potential match. Because no match is found - since 'match `^about/index/`' is a more granular regular expression - Django triggers the first action to send control to the `about.html` template, which was the only regular expression match.

As you can see, Listing 2-1 produces what can be said to be expected behavior. But now let's invert the order of the url regular expressions, as shown in Listing 2-2, and break down why declaring more granular regular expressions toward the bottom is the wrong way to declare Django url regular expressions.

Listing 2-2. **Wrong precedence for Django url regular expressions**

```
from django.views.generic import TemplateVieww

urlpatterns = [
    url(r'^about/',TemplateView.as_view(template_name='about.html')),
    url(r'^about/index/',TemplateView.as_view(template_name='index.html'
]
```

The issue in Listing 2-2 comes when a request is made for the url `/about/index/`. Initially Django matches the last regular expression, which says 'match `^about/index/`'. However, Django continues inspecting the regular expressions and reaches 'match `^about/`' which is a broader match to the request url `/about/index/`, but nevertheless a match! Therefore Django triggers this action and sends control to the `about.html` template, instead of what was likely expected to be the `index.html` template from the first match.

EXACT URL PATTERNS : FORGOING BROAD MATCHING

In the past section, I intentionally used regular expressions that allowed broad url matching. In my experience, as a Django project grows you'll eventually face the need to use this type of url regular expression – but more on why this is so, shortly.

As it turns out, it's possible to use exact url regular expressions. Exact url regular expressions remove any ambiguity introduced by the order in which Django url regular expression are declared.

Let's rework the url regular expressions from Listing 2-2 and make them exact regular expressions so their order doesn't matter. Listing 2-3 illustrates exact regular expressions on basis of those in Listing 2-2.

Listing 2-3. **Exact regular expressions, where url order doesn't matter**

```
from django.views.generic import TemplateVieww

urlpatterns = [
    url(r'^about/$',TemplateView.as_view(template_name='about.html')),
    url(r'^about/index/$',TemplateView.as_view(template_name='index.html
]
```

Notice the regular expressions in Listing 2-3 end with the `$` character. This is the regular expression symbol for end of line, which means the regular expression urls only match an exact pattern.

For example, if Django receives a request for the url `/about/index/` it will only match the last regular expression in Listing 2-3, which says 'match `^about/index/$`'. However, it won't match the higher-up `^/about/$` regular expression because this regular expression says match `about/` *exactly* with nothing else after, since the `$` indicates the end of the pattern.

However, as useful as the `$` character is to make stricter url regular expressions, it's important you analyze its behavior. If you plan to use url Search Engine Optimization (SEO) , A/B testing techniques , or simply want to allow multiple urls to run the same action, stricter regular expressions with `$` eventually require more work.

For example, if you start to use urls like `/about/index/`, `/about/email/`,`/about/address/` and they all use the same template or view for processing, exact regular expressions just make the amount of urls you declare larger. Similarly, if you use A/B testing or SEO where lengthier variations of the same url are processed in the same way (e.g., `/about/landing/a/`, `/about/landing/b/`, `/about/the+coffeehouse+in+san+diego/`) broad url matching is much simpler than declaring exact url patterns.

In the end, whether you opt to use exact url regular expression ending in `$`, I would still recommend you maintain the practice of keeping finergrained url regulars at the top and broader ones at the bottom, as this avoids the unexpected behaviors described in Listing 2-2 when more than one regular expression matches a url request.

COMMON URL PATTERNS

Although url regular expressions can have limitless variations – making it next to impossible to describe each possibility – I'll provide examples on some of the most common url patterns you're more likely to use. Table 2-1 shows individual regular expression characters for Django urls and Table 2-2 shows a series of more concrete examples with url patterns.

Table 2-1. **Regular expression syntax for Django urls: Symbol (Meaning)**

^ (Start of url)	$ (End of url)	\ (Escape for interpreted values)	| (Or)
+ (1 or more occurrences)	? (0 or 1 occurrences)	{n} (n occurrences)	{n,m} (Betwee and m occurrer
[] (Character grouping)	(?P<name>____) (Capture occurrence that matches regexp ____ and assign it to name	. (Any character)	\d+ (On more dig Note esc without escape matches literally.
\D+ (One or more non-digits).Note escape, without escape matches 'D+' literally]	[a-zA-Z0-9_]+ (One or more word characters, letter lower or uppercase, number, or underscore)	\w+ (One or more word characters, equivalent to [a-zA-Z0-9_]). Note escape, without escape matches 'w+' literally].	[-@\w]+ (One or more wo characte dash. or sign). N no escap for \w si it's enclo in brack (i.e., a grouping

Table 2-2. **Common Django url patterns and their regular expressions, with samples**

Url regular expression	Description	Sa

Url regular expression	Description	Sa
url(r'^$',.....)	Empty string (Home page)	Ma htt
url(r'^stores/',.....)	Any trailing characters	Ma ht ht (htt
url(r'^about/contact/$',.....)	Exact, no trailing characters	Ma (htt Do
url(r'^stores/\d+/',.....)	Number	Ma ht ht Do (htt
url(r'^drinks/\D+/',.....)	Non-digits	Ma (htt Do (htt
url(r'^drinks/mocha|espresso/',.....)	Word options, any trailing characters	Ma (htt ht (htt ht Do (htt

Url regular expression	Description	Sa
url(r'^drinks/mocha$\|espresso/$',.....)	Word options exact, no trailing characters	Ma (htt Do (htt Ma (htt Do (htt
url(r'^stores/\w+/',.....)	Word characters (Any letter lower or uppercase, number, or underscore)	Ma (htt ht ht Do ht
url(r'^stores/[-\w]+/',.....)	Word characters or dash	Ma ht
url(r'^state/[A-Z]{2}/',.....)	Two uppercase letters	Ma ht Do ht

Django Urls Don'T Inspect Url Query Strings

On certain urls - those made by HTTP GET requests, common in HTML forms or REST services - parameters are added to urls with `?` followed by `parameter_name=parameter_value` separated by `&` (e.g., `/drinks/mocha/?type=cold&size=large`). These set of values are known as query strings and Django ignores them for the purpose of url pattern matching.

If you need to make use of these values as url parameters – a topic explored in the next section – you can access these values in Django view methods through the request reference. Another alternative is to change the url structure to accommodate regular expressions (e.g.,

/drinks/mocha/cold/large/ instead of /drinks/mocha/?type=cold&size=large).

Url Parameters, Extra Options, and Query Strings

You just learned how to use a wide variety of regular expressions to create urls for your Django applications. However, if you look back at Listings 2-1, 2-2, and 2-3, you'll notice the information provided on the urls is discarded.

Sometimes it's helpful or even necessary to pass url information to the processing construct as a parameter. For example, if you have several urls like `/drinks/mocha/`, `/drinks/espresso/`, and `/drinks/latte/`, the last part of the url represents a drink name. Therefore it can be helpful or necessary to relay this url information to the processing template to display it or use it in some other way in a view (e.g., query a database). To relay this information the url needs to treat this information as a parameter.

To handle url parameters Django uses Python's standard regular expression syntax for named groups. [2] Listing 2-4 shows a url that creates a parameter named `drink_name`.

Listing 2-4. **Django url parameter definition for access in templates**

```
urlpatterns = [
    url(r'^drinks/(?P<drink_name>\D+)/',TemplateView.as_view(template
]
```

Notice the `(?P<drink_name>\D+)` syntax in Listing 2-4. The `?P<>` syntax tells Django to treat this part of the regular expression as a named group and assign the value to a parameter named `drink_name` declared between `<>`. The final piece `\D+` is a regular expression to determine the matching value; in this case the matching value is one or more non-digit characters, as described in Table 2-1.

It's very important you understand a parameter is only captured if the provided value matches the specified regular expression (e.g., \D+ for non-digits). For example, for the url request `/drinks/mocha/` the value `mocha` is assigned to the `drink_name` parameter, but for a url like `/drinks/123/` the regular expression pattern doesn't match - because `123` are digits - so no action is taken.

If a url match occurs in Listing 2-4, the request is sent directly to the template `drinks/index.html`. Django provides access to all parameters defined in this manner through a Django template context variable with the same name. Therefore to access the parameter you would use the parameter name `drink_type` directly in the template. For example, to output the value of the `drink_name` parameter you would use the standard `{{}}` Django template syntax (e.g., `{{drink_name}}`).

In addition to treating parts of a url as parameters, it's also possible to define extra options in the url definition to access them in Django templates as context variables. These extra options are defined inside a dictionary declared as the last part of the url definition.

For example, look at the following modified url Django definition from Listing 2-4:

```
url(r'^drinks/(?P<drink_name>\D+)', TemplateView.as_view(template_nar
```

Notice how a dictionary with key-values is added at the end of the url definition. In this manner, the `onsale` key becomes a url extra option, which is passed to the underlying template as a context variable. Url extra options are accessed like url parameters as template context variables. So to output the `onsale` extra option you would use the `{{onsale}}` syntax.

Next, let's take a look at another variation of url parameters illustrated in Listing 2-5, which sends control to a Django view method.

Listing 2-5. **Django url parameter definition for access in view methods in main urls.py file**

```
# Project main urls.py
from coffeehouse.stores import views as stores_views

urlpatterns = patterns[
    url(r'^stores/(?P<store_id>\d+)/',stores_views.detail),
]
```

Notice the `(?P<store_id>\d+)` syntax in Listing 2-5 is pretty similar to the one in 2-4. The thing that changes is the parameter is now named `store_id` and the regular expression is `\d+` to match digits. So, for example, if a request is made to the url `/stores/1/` the value 1 is assigned to the `store_id` parameter and if a request is made to a url like `/stores/downtown/` the regular expression pattern doesn't match - because `downtown` are letters not digits - so no action is taken.

If a url match occurs for Listing 2-5, the request is sent directly to the Django view method `coffeehouse.stores.views.detail`. Where `coffeehouse.stores` is the package name, `views.py` the file inside the stores app and `detail` the name of the view method. Listing 2-6 illustrates the `detail` view method to access the `store_id` parameter.

Listing 2-6. **Django view method in views.py to access url parameter**

```
from django.shortcuts import render

def detail(request,store_id):
    # Access store_id with 'store_id' variable
    return render(request,'stores/detail.html')
```

Notice in Listing 2-6 how the `detail` method has two arguments. The first argument is a `request` object, which is always the same for all Django view methods. The second argument is the parameter passed by the url. It's important to note the names of url parameters must match the names of the method arguments. In this case, notice in Listing 2-5 the parameter name is `store_id` and in Listing 2-6 the method argument is also named `store_id`.

With access to the url parameter via the view method argument, the method can execute logic with the parameter (e.g., query a database) that can then be passed to a Django template for presentation.

Caution

Django url parameters are *always* treated as strings, irrespective of the regular expression. For example, \d+ catches digits, but a value of one is treated as '1' (String), not 1 (Integer). This is particularly important if you plan to work with url parameters in view methods and do operations that require something other than strings.

Another option available for url parameters handled by view methods is to make them optional, which in turn allows you to leverage the same view method for multiple urls. Parameters can be made optional by assigning a default value to a view method argument. Listing 2-7 shows a new url that calls the same view method (`coffeehouse.stores.views.detail`) but doesn't define a parameter.

Listing 2-7. **Django urls with optional parameters leveraging the same view method**

```
from coffeehouse.stores import views as stores_views

urlpatterns = patterns[
    url(r'^stores/',stores_views.detail),
    url(r'^stores/(?P<store_id>\d+)/',stores_views.detail),
]
```

If you called the url `/stores/` without modifying the `detail` method in Listing 2-6, you would get an error. The error occurs because the `detail` view method expects a `store_id` argument, which isn't provided by the first url. To fix this problem, you can define a default value for the `store_id` in the view method, as illustrated in Listing 2-8.

Listing 2-8. **Django view method in views.py with default value**

```
from django.shortcuts import render

def detail(request,store_id='1'):
    # Access store_id with 'store_id' variable
    return render(request,'stores/detail.html')
```

Notice in Listing 2-8 how the `store_id` argument has the assignment `='1'`. This means the argument will have a default value of `'1'` in case the view method is called without `store_id`. This approach allows you to leverage the same view method to handle multiple urls with optional parameters.

In addition to accessing url parameters inside view methods, it's also possible to access extra options from the url definition. These extra options are defined inside a dictionary declared as the last argument in a url definition. After the view method declaration, you add a dictionary with the key-value pairs you wish to access inside the view method. The following snippet illustrates a modified version of the url statement in Listing 2-7.

```
url(r'^stores/',stores_views.detail,{'location':'headquarters'})
```

In this case, the `location` key becomes a url extra option that's passed as a parameter to the view method. Url extra options are accessed just like url parameters, so to access a url extra option inside a view method you need to modify the method signature to accept an argument with the same name as the url extra option. In this case, the method signature:

```
def detail(request,store_id='1'):
```

needs to change to:

```
def detail(request,store_id='1',location=None):
```

Notice the `location` argument is made optional by assigning a default value of `None`.

Finally, it's also possible to access url parameters separated by `?` And `&` – technically known as a query string – inside Django view methods. These type of parameters can be accessed inside a view method using the `request` object.

Take, for example, the url `/stores/1/?hours=sunday&map=flash`, Listing 2-9 illustrates how to extract the arguments from this url separated by `?` and `&` using `request.GET`.

Listing 2-9. **Django view method extracting url parameters with request.GET**

```
from django.shortcuts import render

def detail(request,store_id='1',location=None):
    # Access store_id param with 'store_id' variable and location param
    # Extract 'hours' or 'map' value appended to url as
    # ?hours=sunday&map=flash
    hours = request.GET.get('hours', '')
    map = request.GET.get('map', '')
    # 'hours' has value 'sunday' or '' if hours not in url
    # 'map' has value 'flash' or '' if map not in url
    return render(request,'stores/detail.html')
```

Listing 2-9 uses the syntax `request.GET.get(<parameter>, '')`. If the parameter is present in `request.GET` it extracts the value and assigns it to a variable for further usage; if the parameter is not present then the parameter variable is assigned a default empty value of `''` – you could equally use `None` or any other default value – as this is part of Python's standard dictionary `get()` method syntax to obtain default values.

This last process is designed to extract parameters from an HTTP GET request; however, Django also supports the syntax `request.POST.get` to extract parameters from an HTTP POST request, which is described in greater detail in the chapter on Django forms and later in this chapter in the section on Django view method requests.

Url Consolidation and Modularization

By default, Django looks up url definitions in the `urls.py` file inside a project's main directory - it's worth mentioning this is on account of the `ROOT_URLCONF` variable in `settings.py`. However, once a project grows beyond a couple of urls, it can become difficult to manage them inside this single file. For example, look at the `urls.py` file illustrated in Listing 2-10.

Listing 2-10. **Django urls.py with no url consolidation**

```
from django.conf.urls import url
from django.views.generic import TemplateView
from coffeehouse.about import views as about_views
from coffeehouse.stores import views as stores_views

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html')),
    url(r'^about/',about_views.index),
    url(r'^about/contact/',about_views.contact),
    url(r'^stores/',stores_views.index),
    url(r'^stores/(?P<store_id>\d+)/',stores_views.detail,{'location':'h
   ]
```

As you can see in Listing 2-10, there are a couple of urls that have redundant roots - `about/` and `stores/`. Grouping these urls separately can be helpful because it keeps common urls in their own files and avoids the difficulties of making changes to one big `urls.py` file.

Listing 2-11 shows an updated version of the `urls.py` file with the `about/` and `stores/` roots placed in separate files.

Listing 2-11. **Django urls.py with include to consolidate urls**

```
from django.conf.urls import include, url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html')),
    url(r'^about/',include('coffeehouse.about.urls')),
    url(r'^stores/',include('coffeehouse.stores.urls'),{'location':'head
   ]
```

Listing 2-11 makes use of the `include` argument to load urls from completely separate files. In this case, `include('coffeehouse.about.urls')` tells Django to load url definitions from the Python module `coffeehouse.about.urls`, which parting from a Django base directory corresponds to the file route `/coffeehouse/about/urls.py`. In this case, I kept using the `urls.py` file name and placed it under the corresponding Django about app directory since it deals with `about/` urls. However, you can use any file name or path you like for url definitions (e.g., `coffeehouse.allmyurl.resturls` to load urls from a file route `/coffeehouse/allmyurls/resturls.py`).

The second include statement in Listing 2-11 works just like the first one, where `include('coffeehouse.stores.urls')` tells Django to load url definitions from the Python module `coffeehouse.stores.urls`. However, notice this second statement appends an additional dictionary as a url extra option, which means all the urls in the include statement will also receive this extra option.

Listing 2-12 illustrates the contents of the file `/coffeehouse/about/urls.py` linked via `include('coffeehouse.about.urls')`.

Listing 2-12. **Django /coffeehouse/about/urls.py loaded via include**

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$',views.index),
    url(r'^contact/$',views.contact),
]
```

A quick look at Listing 2-12 and you can see the structure is pretty similar to the main `urls.py` file; however, there are some minor differences. While the url regular expression `r'^$'` can look like it matches the home page, it isn't. Because the file in Listing 2-12 is linked via `include` in the main `urls.py` file, Django joins the url regular expression with the parent url regular expression. So the first url in Listing 2-12 actually matches `/about/` and the second url in Listing 2-12 actually matches `/about/contact/`. Also because the `urls.py` file in Listing 2-12 is placed alongside the app's `views.py` file, the import statement uses the relative path `from . import views` syntax.

In addition to using the `include` option to reference a separate file with url definitions, the `include` option can also accept url definitions as a

Python list. In essence, this allows you to keep all url definitions in the main `urls.py` file, but give it more modularity . This approach is illustrated in Listing 2-13.

Listing 2-13. **Django urls.py with inline include statements**

```
from django.conf.urls import include, url
from django.views.generic import TemplateView

from coffeehouse.about import views as about_views
from coffeehouse.stores import views as stores_views

store_patterns = [
    url(r'^$',stores_views.index),
    url(r'^(?P<store_id>\d+)/$',stores_views.detail),
]

about_patterns = [
    url(r'^$',about_views.index),
    url(r'^contact/$',about_views.contact),
]

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html')),
    url(r'^about/',include(about_patterns)),
    url(r'^stores/',include(store_patterns),{'location':'headquarters'})
   ]
```

The outcome of the url patterns in Listing 2-13 is the same as Listings 2-11 and 2-12. The difference is Listing 2-13 uses the main `urls.py` file to declare multiple url lists, while Listings 2-11 and 2-12 rely on url lists declared in different files.

Url Naming and Namespaces

A project's internal links or url references (e.g., `<a href='/'>Home Page</a>`) tend to be hard-coded, whether it's in view methods to redirect users to certain locations or in templates to provide adequate user navigation. Hard-coding links can present a serious maintenance problem as a project grows, because it leads to links that are difficult to detect and fix. Django offers a way to name urls so it's easy to reference them in view methods and templates.

The most basic technique to name Django urls is to add the `name` attribute to `url` definitions in `urls.py`. Listing 2-14 shows how to name a project's home page, as well as how to reference this url from a view method or template.

Listing 2-14. **Django url using name**

```
# Definition in urls.py
url(r'^$',TemplateView.as_view(template_name='homepage.html'),name="home

# Definition in view method
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    return HttpResponsePermanentRedirect(reverse('homepage'))

# Definition in template
<a href="{% url 'homepage' %}">Back to home page</a>
```

The url definition in Listing 2-14 uses the regular expression `r'^$'` that translates into `/` or the home page, also known as the root directory. Notice the `name` attribute with the `homepage` value. By assigning the url a `name` you can use this value as a reference in view methods and templates, which means any future changes made to the url regular expression, automatically update all url definitions in view methods and templates.

Next in Listing 2-14 you can see a view method example that redirects control to `reverse('homepage')`. The Django `reverse` method attempts to look up a url definition by the given name - in this case `homepage` - and substitutes it accordingly. Similarly, the link sample `<a href="{% url 'homepage' %}">Back to home page</a>` in Listing 2-14 makes use of the Django `{% url %}` tag, which attempts to look up a url by its first argument - in this case `homepage` - and substitute it accordingly.

This same naming and substitution process is available for more complex url definitions, such as those with parameters. Listing 2-15 shows the process for a url with parameters.

Listing 2-15. **Django url with arguments using name**

```
# Definition in urls.py
url(r'^drinks/(?P<drink_name>\D+)/',TemplateView.as_view(template_name='

# Definition in view method
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    return HttpResponsePermanentRedirect(reverse('drink', args=(drink.na

# Definition in template
<a href="{% url 'drink' drink.name %}">Drink on sale</a>

<a href="{% url 'drink' 'latte' %}">Drink on sale</a>
```

The url definition in Listing 2-15 uses a more complex regular expression with a parameter that translates into urls in the form `/drinks/latte/` or `/drinks/espresso/`. In this case, the url is given the argument name `drink_name`.

Because the url uses a parameter, the syntax for the `reverse` method and `{% url %}` tag are slightly different. The `reverse` method requires the url parameters be provided as a tuple to the `args` variable and the `{% url %}` tag requires the url arguments be provided as a list of values. Notice in Listing 2-15 the parameters can equally be variables or hard-coded values, so long as it matches the url argument regular expression type - which in this case is non-digits.

For url definitions with more than one argument, the approach to using `reverse` and `{% url %}` is identical. For the `reverse` method you pass it a tuple with all the necessary parameters and for the `{% url %}` tag you pass it a list of values.

Caution

Beware of invalid url definitions with reverse and {% url %}. Django always checks at startup that all reverse and {% url %} definitions are valid. This means that if you make an error in a reverse method or {% url %} tag definition - like a typo in the url name or the arguments types don't match the regular expression - the application won't start and throw an HTTP 500 internal error.

The error for this kind of situation is `NoReverseMatch at....Reverse for 'urlname' with arguments '()' and` `keyword arguments '{}' not found. X pattern(s) tried.` If you look at the error stack you'll be able to pinpoint where this is happening and correct it. Just be aware this is a fatal error and if it is not isolated to the view or page where it happens, it will stop the entire application at startup.

Sometimes the use of the `name` attribute by itself is not sufficient to classify urls. What happens if you have two or three index pages? Or if you have two urls that qualify as details, but one is for stores and the other for drinks?

A crude approach would be to use composite names (e.g., drink_details, store_details). However, the use of composite names in this form can lead to difficult-to-remember naming conventions and sloppy hierarchies. A cleaner approach supported by Django is through the `namespace` attribute.

The `namespace` attribute allows a group of urls to be identified with a unique qualifier. Because the `namespace` attribute is associated with a group of urls, it's used in conjunction with the `include` method described earlier to consolidate urls.

Listing 2-16 illustrates a series of url definitions that make use of the namespace attribute with include.

Listing 2-16. **Django urls.py with namespace attribute**

```
# Main urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html'),name="
    url(r'^about/',include('coffeehouse.about.urls',namespace="about")),
    url(r'^stores/',include('coffeehouse.stores.urls',namespace="stores"
]

# About urls.py
from . import views

urlpatterns = [
    url(r'^$',views.index,name="index"),
    url(r'^contact/$',views.contact,name="contact"),
]

# Stores urls.py
from . import views

urlpatterns = [
    url(r'^$',views.index,name="index"),
    url(r'^(?P<store_id>\d+)/$',views.detail,name="detail"),
)

# Definition in view method
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    return HttpResponsePermanentRedirect(reverse('about:index'))

# Definition in template
<a href="{% url 'stores:index' %}">Back to stores index</a>
```

Listing 2-16 starts with a set of `include` definitions typical of a main Django `urls.py` file. Notice both definitions use the `namespace` attribute. Next, you can see the `urls.py` files referenced in the main `urls.py` file that make use of the `name` attribute described in the past example. Notice both the about and stores `urls.py` files have a url with `name='index'`.

To qualify a url name with a namespace you use the syntax `<namespace>:<name>`. As you can see toward the bottom of Listing 2-16, to reference the index in the about `urls.py` you use `about:index` and to reference the index in the stores `urls.py` file you use `stores:index`.

The namespace attribute can also be nested to use the syntax `<namespace1>:<namespace2>:<namespace3>:<name>` to reference urls. Listing 2-17 shows an example of nested namespace attributes.

Listing 2-17. **Django urls.py with nested namespace attribute**

```
# Main urls.py
from django.conf.urls import include, url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html'),name="
    url(r'^stores/',include('coffeehouse.stores.urls',namespace="stores"
]

# Stores urls.py
from . import views

urlpatterns = [
    url(r'^$',views.index,name="index"),
    url(r'^(?P<store_id>\d+)/$',views.detail,name="detail"),
    url(r'^(?P<store_id>\d+)/about/',include('coffeehouse.about.urls',na
]

# About urls.py
from . import views

urlpatterns = [
    url(r'^$',views.index,name="index"),
    url(r'^contact/$',views.contact,name="contact"),
]

# Definition in view method
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    return HttpResponsePermanentRedirect(reverse('stores:about:index', a

# Definition in template
<a href="{% url 'stores:about:index' store.id %}">See about for {{store.
```

The url structure in Listing 2-17 differs from Listing 2-16 in that it creates about urls for each store (e.g., `/stores/1/about/`) instead of having a generic about url (e.g., `/about/`). At the top of Listing 2-17 we use `namespace="stores"` to qualify all urls in the stores `urls.py` file.

Next, inside the stores `urls.py` file notice there's another `include` element with `namespace="about"` to qualify all urls in the about `urls.py`. And finally inside the about `urls.py` file, there are urls that just use the `name` attribute. In the last part of Listing 2-17, you can see how nested namespaces are used with the `reverse` method and `{% url %}` tag using a `:` to separate namespaces.

In 99% of Django urls you can use the `name` and `namespace` parameters just as they been described. However, the `namespace` parameter takes on special meaning when you deploy multiple instances of the same Django app in the same project.

Since Django apps are self-contained units with url definitions, it raises an edge case even if Django apps use url namespaces. What happens if a Django app uses namespace X, but you want to deploy the app two or three times in the same project? How do you reference urls in each app, given they're all written to use namespace X? This is where the term *instance* namespace and the `app_name` attribute come into the picture.

Let's walk through a scenario that uses multiple instances of the same Django app to illustrate this edge case associated with url namespaces. Let's say you develop a Django app called banners to display advertisements. The banners app is built in such a way that it has to run on different urls (e.g., `/coffeebanners/`,`/teabanners/`,`/foodbanners/`) to simplify the selection of banners. In essence, you are required to run multiple instances of the banners app in the same project, each one on different urls.

So what's the problem of multiple app instances and url naming? It has to do with using named urls that need to change dynamically based on the current app instance. This issue is easiest to understand with an example, so let's jump to the example in Listing 2-18.

Listing 2-18. **Django urls.py with multiple instances of the same app**

```
# Main urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html'),name="
    url(r'^coffeebanners/',include('coffeehouse.banners.urls',namespace=
    url(r'^teabanners/',include('coffeehouse.banners.urls',namespace="te
    url(r'^foodbanners/',include('coffeehouse.banners.urls',namespace="f
]

# Banners urls.py
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$',views.index,name="index"),
]

# Definition in view method
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    return HttpResponsePermanentRedirect(reverse('coffee-banners:index')
    return HttpResponsePermanentRedirect(reverse('tea-banners:index'))
    return HttpResponsePermanentRedirect(reverse('food-banners:index'))

# Definition in template
<a href="{% url 'coffee-banners:index' %}">Coffee banners</a>
<a href="{% url 'tea-banners:index' %}">Tea banners</a>
<a href="{% url 'food-banners:index' %}">Food banners</a>
```

In Listing 2-18 you can see we have three urls that point to the same `coffeehouse.banners.urls` file and each has its own unique namespace. Next, let's take a look at the various `reverse` method and `{% url %}` tag examples in Listing 2-18.

Both the `reverse` method and `{% url %}` tag examples in Listing 2-18 resolve to the three different url names using the `<namespace>:<name>` syntax. So you can effectively deploy multiple instances of the same Django app using just `namespace` and `name`.

However, by relying on just `namespace` and `name` the resolved url names cannot adapt dynamically to the different app instances, which is an edge case associated with *internal* app logic that must be included to support multiple instances of a Django app. Now let's take a look at both a view and template scenario that illustrates this scenario and how the `app_name` attribute solves this problem.

Suppose inside the banners app you want to redirect control to the app's main index url (e.g., due to an exception). Now put on an app designer hat, how would you resolve this problem? As an app designer you don't even know about the coffee-banners, tea-banners or food-banners namespaces, as these are deployment namespaces. How would you internally integrate a redirect in the app that adapts to multiple instances of the app being deployed? This is the purpose of the `app_name` parameter.

Listing 2-19 illustrates how to leverage the `app_name` attribute to dynamically determine where to make a redirect.

Listing 2-19. **Django redirect that leverages app_name to determine url**

```
# Main urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^$',TemplateView.as_view(template_name='homepage.html'),name="
    url(r'^coffeebanners/',include('coffeehouse.banners.urls',namespace=
    url(r'^teabanners/',include('coffeehouse.banners.urls',namespace="te
    url(r'^foodbanners/',include('coffeehouse.banners.urls',namespace="f
]

# Banners urls.py
from django.conf.urls import url
from . import views

app_name = 'banners_adverts'
urlpatterns = [
    url(r'^$',views.index,name="index"),
]

# Logic inside Banners app
from django.http import HttpResponsePermanentRedirect
from django.core.urlresolvers import reverse

def method(request):
    ....
    try:
       ...
    except:
       return HttpResponsePermanentRedirect(reverse('banners_adverts:ind
```

Notice the `urls.py` file in Listing 2-19 of the banners app sets the `app_name` attribute before declaring the `urlpatterns` value. Next, notice the `reverse` method in Listing 2-19 uses the `banners_adverts:index` value, where `banners_adverts` represents the `app_name`. This is an important convention, because Django relies on the same syntax to search for `app_name` or `namespace` matches.

So to what url do you think `banners_adverts:index` resolves to? It all depends on where the navigation takes place, it's dynamic! If a user is navigating through the coffee-banners app instance (i.e., url `coffeebanners`) then Django resolves `banners_adverts:index` to the coffee-banners instance index, if a user is navigating through the tea-banners app instance (i.e., url `teabanners`) then Django resolves `banners_adverts:index` to the tea-banners instance index, and so on for any other number of instances. In case a user is navigating outside of a banners app instance (i.e., there is no app instance) then Django defaults to resolving `banners_adverts:index` to the last defined instance in `urls.py`, which would be food-banners.

In this manner and based on where the request path instance a user is coming from (e.g., if the user is on a path with `/coffeebanners/` or `/teabanners/`), the reverse method resolves `banners_adverts:index` dynamically to one of the three url app instances vs. hard-coding specific url namespaces as shown in Listing 2-18.

Now let's assume the banners app has an *internal* template with a link to the app's main `index` url. Similarly, how would you generate this link in the template to take into account the possibility of multiple app instances? Relying on the same `app_name` parameter solves this problem for the template link illustrated in Listing 2-20.

Listing 2-20. **Django template link that leverages app_name to determine url**

```
# template banners/index.html
<a href="{% url 'banners_adverts:index' %}">{% url 'banners_adverts::
```

Notice the `{% url %}` tag in Listing 2-20 points to `banners_adverts:index`. The resolution process for the `banners_adverts:index` is the same outlined in the previous method example that uses the `reverse` method.

If a user is navigating through the coffee-banners app instance (i.e., url `coffeebanners`) then Django resolves `banners_adverts:index` to the coffee-banners instance index, if a user is navigating through the tea-banners app instance (i.e., url `teabanners`) then Django resolves `banners_adverts:index` to the tea-banners instance index, and so on for any other number of instances. In case a user is navigating outside of a banners app instance (i.e., there is no app instance) then Django defaults to resolving `banners_adverts:index` to the last defined instance in `urls.py` that would be `food-banners`.

As you can see, the `app_name` attribute's purpose is to give Django app designers an internal mechanism by which to integrate logic for named urls that dynamically adapt to multiple instances of the same app. For this reason, it's not as widely used for url naming and can be generally foregone in most cases in favor of just using the `namespace` and `name` attributes.

View Method Requests

So far you've worked with Django view methods and their input – a `request` object and parameters – as well as their output, consisting of generating a direct response or relying on a template to generate a response. However, now it's time to take a deeper look at what's available in view method requests and the various alternatives to generate view method responses.

The `request` reference you've placed unquestionably in view methods up to this point, is an instance of the `django.http.request.HttpRequest` class. [3] This `request` object contains information set by entities present before a view method: a user's web browser, the web server that runs the application, or a Django middleware class configured on the application.

The following list shows some of the most common attributes and methods available in a `request` reference:

- `request.method`.- Contains the HTTP method used for the request (e.g., `GET`, `POST`).

- `request.GET` or `request.POST`.- Contains parameters added as part of a GET or POST request, respectively. Parameters are enclosed as a `django.http.request.QueryDict` [4] instance.
    - `request.POST.get('name',default=None)`.- Gets the value of the `name` parameter in a POST request or gets `None` if the parameter is not present. Note `default` can be overridden with a custom value.
    - `request.GET.getlist('drink',default=None)`.- Gets a list of values for the `drink` parameter in a GET request or gets an empty list `None` if the parameter is not present. Note `default` can be overridden with a custom value.
- `request.META`.- Contains HTTP headers added by browsers or a web server as part of the request. Parameters are enclosed in a standard Python dictionary where keys are the HTTP header names – in uppercase and underscore (e.g., `Content-Length` as key `CONTENT_LENGTH`).
    - `request.META['REMOTE_ADDR']`.- Gets a user's remote IP address.
- `request.user`.- Contains information about a Django user (e.g., username, email) linked to the request. Note `user` refers to the user in the `django.contrib.auth` package and is set via Django middleware, described later in this chapter.

As you can attest from this brief list, the `request` reference contains a lot of actionable information to fulfill business logic (e.g., you can respond with certain content based on geolocation information from a user's IP address). There are well over 50 `request` options available between `django.http.request.HttpRequest` and `django.http.request.QueryDict` attributes and methods, all of which are explained in parts of the book where they're pertinent – however you can review the full extent of `request` options in the footnote links in the previous page.

Once you're done extracting information from a `request` reference and doing related business logic with it (e.g., querying a database, fetching data from a third-party REST service), you then need to set up data in a view method to send it out as part of a response.

To set up data in a Django view method, you first need to declare it or extract it inside the method body. You can declare strings, numbers, lists, tuples, dictionaries, or any other Python data structure.

Once you declare or extract the data inside a view method, you create a dictionary to make the data accessible on Django templates. The dictionary keys represent the reference names for the template, while the values are the data structures themselves. Listing 2-21 illustrates a view method that declares multiple data structures and passes them to a Django template.

Listing 2-21. **Set up dictionary in Django view method for access in template**

```
from django.shortcuts import render

def detail(request,store_id='1',location=None):
    # Create fixed data structures to pass to template
    # data could equally come from database queries
    # web services or social APIs
    STORE_NAME = 'Downtown'
    store_address = {'street':'Main #385','city':'San Diego','state':'CA
    store_amenities = ['WiFi','A/C']
    store_menu = ((0,''),(1,'Drinks'),(2,'Food'))
    values_for_template = {'store_name':STORE_NAME, 'store_address':stor
    return render(request,'stores/detail.html', values_for_template)
```

Notice in Listing 2-21 how the `render` method includes the `values_for_template` dictionary. In previous examples, the `render` method just included the `request` object and a template to handle the request. In Listing 2-21, a dictionary is passed as the last `render` argument. By specifying a dictionary as the last argument, the dictionary becomes available to the template - which in this case is `stores/detail.html`.

Tip

If you plan to access the same data on multiple templates, instead of declaring it on multiple views, you can use a context processor to declare it once and make it accessible on all project templates. The next chapter on Django templates discusses this topic.

The dictionary in Listing 2-21 contains keys and values that are data structures declared in the method body. The dictionary keys become references to access the values inside Django templates.

Output View Method Dictionary in Django Templates

Although the next chapter covers Django templates in depth, the following snippet shows how to output the dictionary values in Listing 2-21 using the {{}} syntax.

```
<h4>{{store_name}} store</h4>
<p>{{store_address.street}}</p>
<p>{{store_address.city}},{{store_address.state}}<
<hr/>
<p>We offer: {{store_amenities.0}} and {{store_ame
<p>Menu includes : {{store_menu.1.1}} and {{store_
```

The first declaration `{{store_name}}` uses the stand-alone key to display the `Downtown` value. The other access declarations use dot(`.`) notation because the values themselves are composite data structures.

The `store_address` key contains a dictionary, so to access the internal dictionary values you use the internal dictionary key separated by a dot(`.`). `store_address.street` displays the street value, `store_address.city` displays the city value, and `store_address.state` displays the state value.

The store_amenities key contains a list that uses a similar dot(.) notation to access internal values. However, since Python lists don't have keys you use the list index number. store_amenities.0 displays the first item in list store_amenities and store_amenities.1 displays the second item in list store_amenities.

The `store_menu key` contains a tuple of tuples that also requires a number on account of the lack of keys. `{{store_menu.1.1}}` displays the second tuple value of the second tuple value of `store_menu` and `{{store_menu.2.1}}` displays the second tuple value of the third tuple of `store_menu`.

View Method Responses

The `render()` method to generate view method responses you've used up to this point is actually a shortcut. You can see toward the top of Listing 2-21, the `render()` method is part of the `django.shortcuts` package.

This means there are other alternatives to the `render()` method to generate a view response, albeit the `render()` method is the most common technique. For starters, there are three similar variations to generate view method responses with data backed by a template, as illustrated in Listing 2-22.

Listing 2-22. **Django view method response alternatives**

```
# Option 1)
from django.shortcuts import render

def detail(request,store_id='1',location=None):
    ...
    return render(request,'stores/detail.html', values_for_template)

# Option 2)
from django.template.response import TemplateResponse

def detail(request,store_id='1',location=None):
    ...
    return TemplateResponse(request, 'stores/detail.html', values_for_te

# Option 3)
from django.http import HttpResponse
from django.template import loader, Context

def detail(request,store_id='1',location=None):
     ...
     response = HttpResponse()
     t = loader.get_template('stores/detail.html')
     c = Context(values_for_template)
     return response.write(t.render(c))
```

The first option in Listing 2-22 is the `django.shortcuts.render()` method that shows three arguments to generate a response: the (required) `request` reference, a (required) template route and an (optional) dictionary – also known as the context – with data to pass to the template.

There are three more (optional) arguments for the `render()` method that are not shown in Listing 2-22: `content_type` that sets the HTTP `Content-Type` header for the response and which defaults to `DEFAULT_CONTENT_TYPE` parameter in `settings.py`, which in itself defaults to `text/html`; `status` that sets the HTTP `Status` code for the response that defaults to `200`; and `using` to specify the template engine – either `jinja2` or `django` – to generate the response. The next section on HTTP handling for the `render()` method describes how to use

`content_type & status`, while Chapters 3 and 4 talk about

Django and Jinja template engines.

The second option in Listing 2-22 is the `django.template.response.TemplateResponse()` class, which in terms of input is nearly identical to the `render()` method. The difference between the two variations is that `TemplateResponse()` can alter a response once a view method is finished (e.g., via middleware), where as the `render()` method is considered the last step in the life cycle after a view method finishes. You should use `TemplateResponse()` when you foresee the need to modify view method responses in multiple view methods after they finish their work, a technique that's discussed in a later section in this chapter on view method middleware.

There are four more (optional) arguments for the `TemplateResponse()` class that are not shown in Listing 2-22: `content_type` that defaults to `text/html`; `status` that defaults to `200`; `charset` that sets the response encoding from the HTTP `Content-Type` header or `DEFAULT_CHARSET` in `settings.py` that in itself defaults to `utf-8`; and `using` to indicate the template engine – either `jinja2` or `django` – to generate the response.

The third option in Listing 2-22 represents the longest, albeit the most flexible response creation process. This process first creates a raw `HTTPResponse` instance, then loads a template with the `django.template.loader.get_template()` method, creates a `Context()` class to load values into the template, and finally writes a rendered template with its context to the `HTTPResponse` instance. Although this is the longest of the three options, it's the preferred choice when a view method response requires advanced options. The upcoming section on built-in response shortcuts for inline and streamed content, has more details on `HTTPResponse` response types.

RESPONSE OPTIONS FOR HTTP STATUS AND CONTENT-TYPE HEADERS

Browsers set HTTP headers in requests to tell applications to take into account certain characteristics for processing. Similarly, applications set HTTP headers in responses to tell browsers to take into account certain characteristics for the content being sent out. Among the most important HTTP headers set by applications like Django are `Status` and `Content-Type`.

The HTTP `Status` header is a three-digit code number to indicate the response status for a given request. Examples of `Status` values are `200`, which is the standard response for successful HTTP requests and `404`, which is used to indicate a requested resource could not be found. The HTTP `Content-Type` header is a MIME (Multipurpose Internet Mail Extensions) type string to indicate the type of content in a response. Examples of `Content-Type` values are `text/html`, which is the standard for an HTML content response and `image/gif`, which is used to indicate a response is a GIF image.

By default and unless there's an error, all Django view methods that create a response with `django.shortcuts.render()`, a `TemplateResponse()` class, or `HttpResponse()` class – illustrated in Listing 2-22 – create a response with the HTTP `Status` value set to `200` and the HTTP `Content-Type` set to `text/html`. Although these default values are the most common, if you want to send a different kind of response (e.g., an error or non-HTML content) it's necessary to alter these values.

Overriding HTTP `Status` and `Content-Type` header values for any of the three options in Listing 2-22 is as simple as providing the additional arguments `status` and/or `content_type`. Listing 2-23 illustrates various examples of this process.

Listing 2-23. **HTTP Content-type and HTTP Status for Django view method responses**

```
from django.shortcuts import render

# No method body(s) and only render() example provided for simplicity

# Returns content type text/plain, with default HTTP 200
```

```
    return render(request,'stores/menu.csv', values_for_template, content_ty

    # Returns HTTP 404, wtih default text/html
    # NOTE: Django has a built-in shortcut & template 404 response, describe
    return render(request,'custom/notfound.html',status=404)

    # Returns HTTP 500, wtih default text/html
    # NOTE: Django has a built-in shortcut & template 500 response, describe
    return render(request,'custom/internalerror.html',status=500)

    # Returns content type application/json, with default HTTP 200
    # NOTE: Django has a built-in shortcut JSON response, described in the n
    return render(request,'stores/menu.json', values_for_template, content_t
```

The first example in Listing 2-23 is designed to return a response with plain text content. Notice the `render` method `content_type` argument. The second and third examples in Listing 2-23 set the HTTP `Status` code to `404` and `500`. Because the HTTP Status `404` code is used for resources that are not found, the `render` method uses a special template for this purpose. Similarly, because the HTTP `Status 500` code is used to indicate an error, the `render` method also uses a special template for this purpose.

Tip

Django has built-in shortcuts and templates to deal with HTTP `Status` codes `404` and `500`, as well as a JSON short-cut response, all of which are described in the next section and that you can use instead of the examples in Listing 2-23.

The fourth and last example in Listing 2-23 is designed to return a response with JavaScript Object Notation(JSON) content. The HTTP `Content-Type application/json` is a common requirement for requests made by browsers that consume JavaScript data via Asynchronous JavaScript (AJAX).

BUILT-IN RESPONSE SHORTCUTS AND TEMPLATES FOR COMMON HTTP STATUS: 404 (NOT FOUND), 500 (INTERNAL SERVER ERROR), 400 (BAD REQUEST), AND 403 (FORBIDDEN)

Although Django automatically triggers an HTTP 404 `Status` (Not Found) response when a page is not found and also triggers an HTTP 500 `Status` (Internal Server Error) response when an unhandled exception is thrown in a view, it has built-in shortcuts and templates that are meant to be used explicitly in Django views when you know end users should get them. Table 2-3 illustrates the different shortcuts to trigger certain HTTP status responses.

Table 2-3. **Django shortcut exceptions to trigger HTTP statuses**

HTTP status code	Python code sample
404 (Not Found)	from django.http import Http404 raise Http404

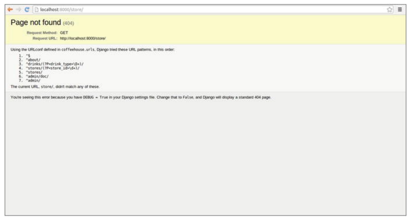
HTTP status code	Python code sample
500 (Internal Server Error)	raise Exception
400 (Bad Request)	from django.core.exceptions import SuspiciousOperation raise SuspiciousOperation
403 (Forbidden)	from django.core.exceptions import PermissionDenied raise PermissionDenied

**Django automatically handles not found pages raising HTTP 404 and unhandled exceptions raising HTTP 500*

As you can see in the examples in Table 2-3, the shortcut syntax is straightforward. For example, you can make evaluations in a Django view like `if article_id < 100:` or `if unpayed_subscription:` and based on the result throw exceptions from Table 2-3 so end users get the proper HTTP status response.

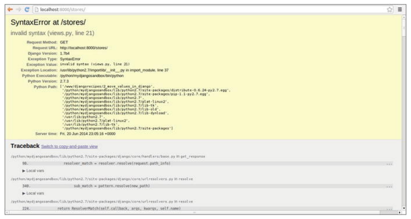
So what is the actual content sent in a response besides the HTTP status when an exception from Table 2-3 is triggered? The default for HTTP `400` (Bad Request) and HTTP `403` (Forbidden) is a single line HTML page that says “`Bad Request (400)`” and “`403 Forbidden`”, respectively. For HTTP `404` (Not Found) and HTTP `500` (Internal Server Error), it depends on the `DEBUG` value in `settings.py`.

If a Django project has `DEBUG=True` in `settings.py`, HTTP `404` (Not Found) generates a page with the available urls - as illustrated in Figure 2-1 - and HTTP `500` (Internal Server Error) generates a page with the detailed error - as illustrated in Figure 2-2. If a Django project has `DEBUG=False` in `settings.py`, HTTP `404` (Not Found) generates a single line HTML page that says “`Not Found. The requested URL <url_location> was not found on this server.`” and HTTP `500` (Internal Server Error) generates a single line HTML page that says “`A server error occurred. Please contact the administrator`".



```
localhost:8000/store/

Page not found (404)

    Request Method: GET
       Request URL: http://localhost:8000/store/

Using the URLconf defined in coffeehouse.urls, Django tried these URL patterns, in this order:

    1. ^$
    2. ^about/
    3. ^drinks/(?P<drink_type>\D+)/
    4. ^stores/(?P<store_id>\d+)/
    5. ^stores/
    6. ^admin/doc/
    7. ^admin/

The current URL, store/, didn't match any of these.

You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.
```

Figure 2-1. HTTP 404 for Django project when DEBUG=True



```
localhost:8000/stores/

SyntaxError at /stores/
invalid syntax (views.py, line 21)

       Request Method: GET
          Request URL: http://localhost:8000/stores/
       Django Version: 1.7b4
       Exception Type: SyntaxError
      Exception Value: invalid syntax (views.py, line 21)
   Exception Location: /usr/lib/python2.7/importlib/__init__.py in import_module, line 37
    Python Executable: /python/mydjangosandbox/bin/python
       Python Version: 2.7.3
          Python Path: ['/www/djangorecipes/2_move_values_in_django',
                        '/python/mydjangosandbox/lib/python2.7/site-packages/distribute-0.6.24-py2.7.egg',
                        '/python/mydjangosandbox/lib/python2.7/site-packages/pip-1.1-py2.7.egg',
                        '/python/mydjangosandbox/lib/python2.7',
                        '/python/mydjangosandbox/lib/python2.7/plat-linux2',
                        '/python/mydjangosandbox/lib/python2.7/lib-tk',
                        '/python/mydjangosandbox/lib/python2.7/lib-old',
                        '/python/mydjangosandbox/lib/python2.7/lib-dynload',
                        '/usr/lib/python2.7',
                        '/usr/lib/python2.7/plat-linux2',
                        '/usr/lib/python2.7/lib-tk',
                        '/python/mydjangosandbox/lib/python2.7/site-packages']
          Server time: Fri, 20 Jun 2014 23:05:16 +0000

Traceback Switch to copy-and-paste view

/python/mydjangosandbox/lib/python2.7/site-packages/django/core/handlers/base.py in get_response
    98.                 resolver_match = resolver.resolve(request.path_info)                ...
    ▶ Local vars
/python/mydjangosandbox/lib/python2.7/site-packages/django/core/urlresolvers.py in resolve
    340.                    sub_match = pattern.resolve(new_path)                           ...
    ▶ Local vars
/python/mydjangosandbox/lib/python2.7/site-packages/django/core/urlresolvers.py in resolve
    224.            return ResolverMatch(self.callback, args, kwargs, self.name)            ...
```

Figure 2-2. HTTP 500 for Django project when DEBUG=True

It's also possible to override the default response page for all the previous HTTP codes with custom templates. To use a custom response page, you need to create a template with the desired HTTP code and `.html` extension. For example, for HTTP `403` you would create the `403.html` template and for HTTP `500` you would create the `500.html` template. All these custom HTTP response templates need to be placed in a folder defined in the `DIRS` list of the `TEMPLATES` variable so Django finds them before it uses the default HTTP response templates.

Caution

Custom 404.html and 500.html pages only work when DEBUG=False.

If `DEBUG=True`, it doesn't matter if you have `404.html` or `500.html` templates in the right location, Django uses the default response behavior illustrated in Figure 2-1 and Figure 2-2, respectively. You need to set `DEBUG=False` for the custom `404.html` and `500.html` templates to work.

On certain occasions, using custom HTTP response templates may not be enough. For example, if you want to add context data to a custom template that handles an HTTP response, you need to customize the built-in Django HTTP view methods themselves, because there's no other way to pass data into this type of template. To customize the built-in Django HTTP view methods you need to declare special handlers in a project's `urls.py` file. Listing 2-24 illustrates the `urls.py` file with custom handlers for Django's built-in HTTP `Status` view methods.

Listing 2-24. **Override built-in Django HTTP Status view methods in urls.py**

```
# Overrides the default 400 handler django.views.defaults.bad_request
handler400 = 'coffeehouse.utils.views.bad_request'
# Overrides the default 403 handler django.views.defaults.permission_den
handler403 = 'coffeehouse.utils.views.permission_denied'
# Overrides the default 404 handler django.views.defaults.page_not_found
handler404 = 'coffeehouse.utils.views.page_not_found'
```

```
# Overrides the default 500 handler django.views.defaults.server_error
handler500 = 'coffeehouse.utils.views.server_error'

urlpatterns = [....
]
```

Caution

If DEBUG=True, the handler404 and handler500 handlers won't work, Django keeps using the built-in Django HTTP view methods. You need to set DEBUG=False for the handler404 and handler500 handlers to work.

As you can see in Listing 2-24, there are a series of variables in `urls.py` right above the standard `urlpatterns` variable. Each variable in Listing 2-24 represents an HTTP `Status` handler, with its value corresponding to a custom Django view to process requests. For example, `handler400` indicates that all HTTP `400` requests should be handled by the Django view method `coffeehouse.utils.views.bad_request` instead of the default `django.views.defaults.bad_request`. The same approach is taken for HTTP `403` requests using `handler403`, HTTP `404` requests using `handler404` and HTTP 500 requests using `handler500`.

As far as the actual structure of custom Django view methods is concerned, they are identical to any other Django view method. Listing 2-26 shows the structure of the custom view methods used in Listing 2-25.

Listing 2-25. **Custom views to override built-in Django HTTP view methods**

```
from django.shortcuts import render

def page_not_found(request):
    # Dict to pass to template, data could come from DB query
    values_for_template = {}
    return render(request,'404.html',values_for_template,status=404)

def server_error(request):
    # Dict to pass to template, data could come from DB query
    values_for_template = {}
    return render(request,'500.html',values_for_template,status=500)

def bad_request(request):
    # Dict to pass to template, data could come from DB query
    values_for_template = {}
    return render(request,'400.html',values_for_template,status=400)

def permission_denied(request):
    # Dict to pass to template, data could come from DB query
    values_for_template = {}
    return render(request,'403.html',values_for_template,status=403)
```

As you can see in Listing 2-26, the custom HTTP view methods use the same `render` method from `django.shortcuts` as the previous view method examples. The methods point to a template named by the HTTP `Status` code, use a custom data dictionary that becomes accessible on the template and uses the `status` argument to indicate the HTTP status code.

BUILT-IN RESPONSE SHORTCUTS FOR INLINE AND STREAMED CONTENT

All the prior view response examples have worked on the basis of content being structured through a template. However, there can be times when using a template to output a response is unnecessary (e.g., a one-line response that says "Nothing to see here").

Other times it makes no sense for a response to use a template, such is the case for HTTP 301 (Permanent Redirect) or HTTP 302 (Redirect) where the response just requires a redirection url. Table 2-4 illustrates the different shortcuts to trigger HTTP redirects.

Table 2-4. **Django shortcuts for HTTP redirects**

HTTP status code	Python code sample

301 (Permanent Redirect)	from django.http import HttpResponsePermanentRedirect
	
	return HttpResponsePermanentRedirect("/")

302 (Redirect)	from django.http import HttpResponseRedirect
	
	return HttpResponseRedirect("/")

Both samples in Table 2-4 redirect to an application's home page (i.e., "/"). However, you can also set the redirection to any application url or even a full url on a different domain (e.g., http://maps.google.com/ (http://maps.google.com/)).

In addition to response redirection shortcuts, Django also offers a series of response shortcuts where you can add inline responses. Table 2-5 illustrates the various other shortcuts for HTTP status codes with inline content responses.

Table 2-5. **Django shortcuts for inline and streaming content responses**

Purpose or HTTP Status code	Python code sample

304 (NOT MODIFIED)	from django.http import HttpResponseNotModified
	
	return HttpResponseNotModified()*

400 (BAD REQUEST)	from django.http import HttpResponseBadRequest
	
	return HttpResponseBadRequest("<h4>T request doesn't look right</h4>")

Purpose or HTTP Status code	Python code sample

404 (NOT FOUND)	from django.http import HttpResponseNotFound
	
	return HttpResponseNotFound("<h4>Up
	can't find that page</h4>")

403 (FORBIDDEN)	from django.http import HttpResponseForbidden
	
	return HttpResponseForbidden("Can't loo
	anything here",content_type="text/plain"

405 (METHOD NOT ALLOWED)	from django.http import HttpResponseNotAllowed
	
	return HttpResponseNotAllowed("<h4>Method
	allowed</h4>")

410 (GONE)	from django.http import HttpResponseGo
	
	return HttpResponseGone("No longer here",content_type="text/plain")

500 (INTERNAL SERVER ERROR)	from django.http import HttpResponseServerError
	
	return HttpResponseServerError("<h4>U
	that's a mistake on our part, sorry!</h4>"

Inline response that serializes data to JSON (Defaults to HTTP 200 and content type application/json)	from django.http import JsonResponse
	
	data_dict = {'name':'Downtown','address':'Main #385','city':'San Diego','state':'CA'}
	
	return JsonResponse(data_dict)

Purpose or HTTP Status code	Python code sample

Inline response that stream data (Defaults to HTTP 200 and streaming content, which is an iterator of strings)	from django.http import StreamingHttpResponse
	
	return StreamingHttpResponse(large_data_stru

Inline response that stream binary files (Defaults to HTTP 200 and streaming content)	from django.http import FileResponse
	
	return FileResponse(open('Report.pdf','rb

Inline response with any HTTP status code (Defaults to HTTP 200)	from django.http import HttpResponse
	
	return HttpResponse("<h4>Django inline response</h4>")

** The HTTP 304 status code indicates a "Not Modified" response, so you can't send content in the response, it should always be empty.*

As you can see in the samples in Table 2-5, there are multiple shortcuts to generate different HTTP Status responses with inline content and entirely forgo the need to use a template. In addition, you can see the shortcuts in Table 2-5 can also accept the `content_type` argument if the content is something other than HTML (i.e., `content_type=text/html`).

Since non-HTML responses have become quite common in web applications, you can see Table 2-5 also shows three Django built-in response shortcuts to output non-HTML content. The `JsonResponse` class is used to transform an inline response into JavaScript Object Notation (JSON). Because this response converts the payload to a JSON data structure, it automatically sets the content type to `application/json`. The `StreamingHttpResponse` class is designed to stream a response without the need to have the entire payload in-memory, a scenario that's helpful for large payload responses. The `FileResponse` class – a subclass of `StreamingHttpResponse` – is designed to stream binary data (e.g., PDF or image files).

This takes us to the last entry in Table 2-5, the `HttpResponse` class. As it turns out, all the shortcuts in Table 2-5 are customized subclasses of the `HttpResponse` class, which I initially described in Listing 2-22 as one of the most flexible techniques to create view responses.

The `HttpResponse` method is helpful to create responses for HTTP status codes that don't have direct shortcut methods (e.g., HTTP `408` [Request Timeout], HTTP `429` [Too Many Requests]) or to inclusively harness a template to generate inline responses as illustrated in Listing 2-26.

Listing 2-26. **HttpResponse with template and custom CSV file download**

```
from django.http import HttpResponse
from django.utils import timezone
from django.template import loader, Context

response = HttpResponse(content_type='text/csv')
response['Content-Disposition'] = 'attachment; filename=Users_%s.csv' %
t = loader.get_template('dashboard/users_csvexport.html')
c = Context({'users': sorted_users,})
response.write(t.render(c))
return response
```

The `HTTPResponse` object in Listing 2-26 is generated with a `text/csv` content type to advise the requesting party (e.g., browser) that it's about to receive CSV content. Next, the `Content-Disposition` header also tells the requesting party (e.g., browser) to attempt to download the content as a file named `Users_%s.csv` where the `%s` is substituted with the current server date.

Next, using the `loader` module we use the `get_template` method to load the template `users_csvexport.html` that will have a CSV-like structure with data placeholders. Then we create a `Context` object to hold the data that will fill the template, which in this case it's just a single variable named `users`. Next, we call the template's `render` method with the `context` object in order to fill in the template's data placeholders with the data. Finally, the rendered template is written to the `response` object via the `write` method and the `response` object is returned.

The `HttpResponse` class offers over 20 options between attributes and methods, [5] in addition to the `content_type` and `status` parameters.

View Method Middleware

In most circumstances, data in requests and responses is added, removed, or updated in a piecemeal fashion in view methods. However, sometimes it's convenient to apply these changes on all requests and responses.

For example, if you want to access certain data on all view methods, it's easier to use a middleware class to make this data accessible across all requests. Just as if you want to enforce a security check on all responses, it's easier to do so globally with a middleware class.

Since middleware is a rather abstract concept, before I describe the structure of a Django middleware class, I'll walk you through the various built-in Django middleware classes so you can get a firmer understanding of where middleware is good design choice.

BUILT-IN MIDDLEWARE CLASSES

Django comes equipped with a series of middleware classes, some of which are enabled by default on all Django projects. If you open a Django project's `settings.py` file you'll notice the `MIDDLEWARE` variable whose default contents are shown in Listing 2-27.

Listing 2-27. **Default Django middleware classes in MIDDLEWARE**

```
MIDDLEWARE = [
    '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',
]
```

As you can see in Listing 2-27, Django projects in their out-of-the-box state come enabled with seven middleware classes, so all requests and responses are set to run through these seven classes. If you plan to leverage Django's main features, I advise you not to remove any of these default middleware classes. However, you can leave the `MIDDLEWARE` variable empty if you wish; just be aware doing so may break certain Django functionalities.

To give you a better understanding of what the Django middleware classes in Listing 2-27 do and help you make a more informed decision to disable them or not, Table 2-6 describes the functionality for each of these middleware classes.

Table 2-6. **Django default middleware classes and functionality**

Middleware class
django.middleware.security.SecurityMiddleware
django.contrib.sessions.middleware.SessionMiddleware
django.middleware.common.CommonMiddleware
django.middleware.csrf.CsrfViewMiddleware

Middleware class
django.contrib.auth.middleware.AuthenticationMiddleware
django.contrib.messages.middleware.MessageMiddleware
django.middleware.clickjacking.XFrameOptionsMiddleware

As you can see in Table 2-6, although the purpose of the various default middleware classes varies considerably, their functionality applies to features that need to be applied across all requests or responses in a project.

Another important factor of the middleware classes in Table 2-6 is that some are dependent on others. For example, the `AuthenticationMiddleware` class is designed on the assumption it will have access to functionality provided by the `SessionMiddleware` class. Such dependencies are important because it makes the middleware class definition order relevant (i.e., certain middleware classes need to be defined before others in `MIDDLEWARE`), a topic I'll elaborate on more in the next section.

In addition to the default middleware classes presented in Table 2-6, Django also offers other middleware classes. Table 2-7 illustrates the remaining set of Django middleware classes you can leverage in your projects, which can be helpful so you don't have to write middleware classes from scratch.

Table 2-7. **Other Django middleware classes and functionality**

Middleware class

Middleware class
django.middleware.cache.UpdateCacheMiddleware
django.middleware.cache.FetchFromCacheMiddleware
django.middleware.common.BrokenLinkEmailsMiddleware
django.middleware.ExceptionMiddleware

Middleware class
django.middleware.gzip.GZipMiddleware
django.middleware.http.ConditionalGetMiddleware
django.middleware.locale.LocaleMiddleware
django.contrib.sites.middleware.CurrentSiteMiddleware
django.contrib.auth.middleware. PersistentRemoteUserMiddleware

Middleware class
django.contrib.auth.middleware.RemoteUserMiddleware
django.contrib.flatpages.middleware.FlatpageFallbackMiddlewa
django.contrib.redirects.middleware.RedirectFallbackMiddlewa

Now that you know about Django's built-in middleware classes and what they're used for, let's take a look at the structure of middleware classes and their execution process.

MIDDLEWARE STRUCTURE AND EXECUTION PROCESS

A Django middleware class has two required methods and three optional methods that execute at different points of the view request/response life cycle. Listing 2-28 illustrates a sample middleware class with its various parts.

Listing 2-28. **Django middleware class structure**

```
class CoffeehouseMiddleware(object):

    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization on start-up

    def __call__(self, request):
        # Logic executed on a request before the view (and other middlew

        # get_response call triggers next phase
        response = self.get_response(request)

        # Logic executed on response after the view is called.
        # Return response to finish middleware sequence
        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        # Logic executed before a call to view
        # Gives access to the view itself & arguments

    def process_exception(self,request, exception):
        # Logic executed if an exception/error occurs in the view

    def process_template_response(self,request, response):
        # Logic executed after the view is called,
        # ONLY IF view response is TemplateResponse, see listing 2-22
```

In order for view methods to execute the Django middleware class in Listing 2-28, middleware classes must be added to the `MIDDLEWARE` variable in `settings.py`. So for example, if the `CoffeehouseMiddleware` class in Listing 2-28 is stored in a file/module named `middleware.py` under the `coffeehouse/utils/` project folders, you would add the `coffeehouse.utils.middleware.CoffeeMiddleware` statement to the list of `MIDDLEWARE` values in `settings.py`.

Next, I'll describe the two required methods in all Django middleware class shown in Listing 2-28:

- `__init__`.- Used in all Python classes to bootstrap object instances. The `__init__` method in Django middleware classes only gets called once, when the web server backing the Django application starts. The `__init__` method in Django middleware must declare a `get_response` input, which represents a reference to a prior middleware class response. The `get_response` input is assigned to an instance variable – also named `get_response` – which is later used in the main processing logic of the middleware class. The purpose of the `get_response` reference should become clearer shortly when I expand on the Django middleware execution process.
- `__call__`.- Used in all Python classes to call an object instance as a function. The `__call__` method in Django middleware classes is called on every application request. As you can see in Listing 2-28, the `__call__` method declares a `request` input that represents the same `HttpRequest` object used by view methods. The `__call__` method goes through three phases:
    - Before view method call.- Once the `__call__` method is triggered, you get the opportunity to alter the `request` reference *before* it's passed to a view method. If you want to add or modify something in `request` before it gets turned over to a view method, this is the phase to do it in.
    - Trigger view method call.- After you modify (or not) the original `request`, you must turn over control to the view method in order for it to run. This phase is triggered when you pass `request` to the `self.get_response` reference you set in the `__init__` method. This phase effectively says, "I'm done modifying the `request`, go ahead and turn it over to the view method so it can run."
    - Post view method call.- Once a view method finishes, the results are assigned to the `response` reference in `__call__`. In this phase, you have the opportunity to perform logic after

a view method finishes. You exit this phase by simply returning the `response` reference from the view method (i.e., `return response`).

This is the core logic behind every Django middleware class performed by these two required methods. Now let's take a look at the three optional middleware class methods presented in Listing 2-28:

- `process_view`.- The required middleware methods – `__init__` and `__call__` – lack any knowledge about the view method they're working on. The `process_view` method gives you access to a view method and its argument *before* the view method is triggered. If present, the `process_view` middleware method is invoked right after `__call__` and before calling `self.get_response(request)`, which triggers the view method.

- `process_exception`.- If an error occurs in the logic of a view method, the `process_exception` middleware method is invoked to give you the opportunity to perform post-error clean-up logic.

- `process_template_response`.- After the self.get_response(request) is called and a view method finishes, it can be necessary to alter the response itself to perform additional logic on it (e.g., modify the context or template). If present, the `process_template_response` middleware method is invoked after a view method finishes to give you the opportunity to tinker with the response.

Warning

The process_template_response middleware method is only triggered if a view method returns a TemplateResponse. If a view method generates a response with render() the process_template_response is not triggered. See Listing 2-22 for view method responses for more details.

In summary, the execution process for a single middleware class is the following:

1. `__init__` method triggered (On server startup).
2. `__call__` method triggered (On every request).
3. If declared, `process_view()` method triggered.
4. View method starts with `self.get_response(request)` statement in `__call__`.
5. If declared, `process_exception()` method triggered when exception occurs in view.
6. View method finishes.
7. If declared, `process_template_response()` triggered when view returns `TemplateResponse`.

Although it's important to understand the execution process of a single middleware class, a more important aspect is to understand the execution process of multiple middleware classes. As I mentioned at the outset of this section, Django projects are enabled with seven middleware classes shown in Listing 2-27, so the execution of multiple middleware classes is more the norm rather than the exception.

Django middleware classes are executed back to back, but the view method represents an inflection point in their execution order. The execution order for the default middleware classes in Listing 2-27 is the following:

```
Server start-up

__init__ on django.middleware.security.SecurityMiddleware called
__init__ on django.contrib.sessions.middleware.SessionMiddleware called
__init__ on django.middleware.common.CommonMiddleware called
__init__ on django.middleware.csrf.CsrfViewMiddleware called
__init__ on django.contrib.auth.middleware.AuthenticationMiddleware call
__init__ on django.contrib.messages.middleware.MessageMiddleware called
__init__ on django.middleware.clickjacking.XframeOptionsMiddleware calle

request for index() view method

__call__ on django.middleware.security.SecurityMiddleware called
process_view on django.middleware.security.SecurityMiddleware called (if
__call__ on django.contrib.sessions.middleware.SessionMiddleware called
process_view on django.contrib.sessions.middleware.SessionMiddleware cal
__call__ on django.middleware.common.CommonMiddleware called
process_view on django.middleware.common.CommonMiddleware called (if dec
__call__ on django.middleware.csrf.CsrfViewMiddleware called
process_view on django.middleware.csrf.CsrfViewMiddleware called (if dec
__call__ on django.contrib.auth.middleware.AuthenticationMiddleware call
process_view on django.contrib.auth.middleware.AuthenticationMiddleware
__call__ on django.contrib.messages.middleware.MessageMiddleware called
process_view on django.contrib.messages.middleware.MessageMiddleware cal
__call__ on django.middleware.clickjacking.XframeOptionsMiddleware calle
process_view on django.middleware.clickjacking.XframeOptionsMiddleware c

start index() view method logic

if an exception occurs in index() view
process_exception on django.middleware.clickjacking.XframeOptionsMiddlew
process_exception on django.contrib.messages.middleware.MessageMiddlewar
process_exception on django.contrib.auth.middleware.AuthenticationMiddle
process_exception on django.middleware.csrf.CsrfViewMiddleware called (i
process_exception on django.middleware.common.CommonMiddleware called (i
process_exception on django.contrib.sessions.middleware.SessionMiddlewar
process_exception on django.middleware.security.SecurityMiddleware calle

if index() view returns TemplateResponse
process_template_response on django.middleware.clickjacking.XframeOption
process_template_response on django.contrib.messages.middleware.MessageM
process_template_response on django.contrib.auth.middleware.Authenticati
process_template_response on django.middleware.csrf.CsrfViewMiddleware c
process_template_response on django.middleware.common.CommonMiddleware c
process_template_response on django.contrib.sessions.middleware.SessionM
process_template_response on django.middleware.security.SecurityMiddlewa
```

Notice the execution order for middleware classes prior to entering the execution of the view method, follows the declared order (i.e., first declared runs first, last declared last). But once the view method is executed, the middleware execution order is inverted (i.e., last declared runs first, first declared last).

This behavior is similar to a corkscrew, where to get to the center (view method), you move in one direction (1 to 7) and to move out you go in the opposite direction (7 to 1). Therefore the middleware methods `process_exception` and `process_template_response` execute in the opposite order of `__init__`, `__call__` and `process_view`.

Visually the execution process for the default Django middleware classes in Listing 2-27 is illustrated in Figure 2-3.

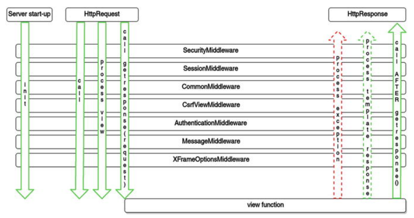
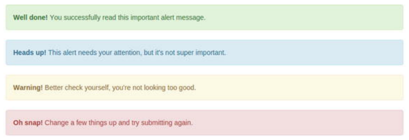


Figure 2-3. Django middleware execution process

Middleware Flash Messages in View Methods

Flash messages are typically used when users perform an action (e.g., submit a form) and it's necessary to tell them if the action was successful or if there was some kind of error. Other times flash messages are used as one-time notifications on web pages to tell users about certain events (e.g., site maintenance or special discounts). Figure 2-4 shows a set of sample flash messages.



Well done! You successfully read this important alert message.

Heads up! This alert needs your attention, but it's not super important.

Warning! Better check yourself, you're not looking too good.

Oh snap! Change a few things up and try submitting again.

Figure 2-4. Web page flash messages

Django Flash Messages Require a Django App, Middleware, and a Template Context Processor

By default, all Django projects are enabled to support flash messages. However, if you tweaked your project's settings.py file you may have inadvertently disabled flash messages.

In order for Django flash messages to work you must ensure the following values are set in settings.py: The variable INSTALLED_APPS has the django.contrib.messages value, the variable MIDDLEWARE has the django.contrib.messages.middleware.MessageMiddleware value, and the context_processors list in OPTIONS of the TEMPLATES variable has the django.contrib.messages.context_processors.messages value.

As you can see in Figure 2-4 there can be different types of flash messages, which are technically known as levels. Django follows the standard Syslog standard severity levels and supports five built-in message levels described in Table 2-8.

Table 2-8. **Django built-in flash messages**

Level Constant	Tag	Value	Purpose

Level Constant	Tag	Value	Purpose
DEBUG	debug	10	Development-related messages that will be ignored (or removed) in a production deployment.
INFO	info	20	Informational messages for the user.
SUCCESS	success	25	An action was successful, for example, “Contact info was sent successfully.”
WARNING	warning	30	A failure did not occur but may be imminent.
ERROR	error	40	An action was not successful or some other failure occurred.

ADD FLASH MESSAGES

Django flash messages are managed on a per request basis and are added in view methods, as this is the best place to determine whether flash messages are warranted. To add messages you use the `django.contrib.messages` package.

There are two techniques to add flash messages with the `django.contrib.messages` package: one is the generic `add_message()` method, and the other is shortcuts methods for the different levels described in Table 2-8. Listing 2-29 illustrates the different techniques .

Listing 2-29. **Techniques to add Django flash messages**

```
from django.contrib import messages

# Generic add_message method
messages.add_message(request, messages.DEBUG, 'The following SQL stateme
messages.add_message(request, messages.INFO, 'All items on this page hav
messages.add_message(request, messages.SUCCESS, 'Email sent successfully
messages.add_message(request, messages.WARNING, 'You will need to change
messages.add_message(request, messages.ERROR, 'We could not process your

# Shortcut level methods
messages.debug(request, 'The following SQL statements were executed: %s'
messages.info(request, 'All items on this page have free shipping.')
messages.success(request, 'Email sent successfully.')
messages.warning(request, 'You will need to change your password in one
messages.error(request, 'We could not process your request at this time.
```

The first set of samples in Listing 2-29 uses the `add_message()` method, where as the second set uses shortcut level methods. Both sets of samples in Listing 2-29 produce the same results.

If you look closely at Listing 2-29 you'll notice both `DEBUG` level messages have the end-of-line comment `# Ignored by default`. The Django messages framework by default processes all messages above the `INFO` level (inclusive), which means `DEBUG` messages – being a lower-level message threshold, as described in Table 2-8 – are ignored even though they might be defined.

You can change the default Django message level threshold to include all message levels or inclusively reduce the default `INFO` threshold. The default message level threshold can be changed in one of two ways: globally (i.e., for the entire project) in `settings.py` with the `MESSAGE_LEVEL` variable as illustrated in Listing 2-30 or on a per request basis with the `set_level` method of the `django.contrib.messages` package as illustrated in Listing 2-31.

Listing 2-30. **Set default Django message level globally in settings.py**

```
# Reduce threshold to DEBUG level in settings.py
from django.contrib.messages import constants as message_constants
MESSAGE_LEVEL = message_constants.DEBUG

# Increase threshold to WARNING level in setting.py
from django.contrib.messages import constants as message_constants
MESSAGE_LEVEL = message_constants.WARNING
```

Listing 2-31. **Set default Django message level on a per request basis**

```
# Reduce threshold to DEBUG level per request
from django.contrib import messages
messages.set_level(request, messages.DEBUG)

# Increase threshold to WARNING level per request
from django.contrib import messages
messages.set_level(request, messages.WARNING)
```

The first `MESSAGE_LEVEL` definition in Listing 2-30 changes the default message level to `DEBUG`, which means all message level definitions get processed, since `DEBUG` is the lowest threshold. The second `MESSAGE_LEVEL` definition in Listing 2-30 changes the default message level to `WARNING`, which means message levels higher than `WARNING` (inclusive) are processed (i.e., `WARNING` and `ERROR`).

The first `set_level` definition in Listing 2-31 changes the default request message level to `DEBUG`, which means all message level definitions get processed, since `DEBUG` is the lowest threshold. The second `set_level` definition in Listing 2-31 changes the default message level to `WARNING`, which means message levels higher than `WARNING` (inclusive) are processed (i.e., `WARNING` and `ERROR`).

If you define both default message level mechanisms at once, the default request message level takes precedence over the default global message level definition (e.g., if you define `messages.set_level(request,` `messages.WARNING)`, message levels above `WARNING` (inclusive) are processed, even if the global `MESSAGE_LEVEL` variable is set to `MESSAGE_LEVEL = message_constants.DEBUG` to include all messages.

In addition to setting up flash messages and knowing about the built-in threshold mechanism that ignores messages from a certain level, it's also important you realize the message definitions in Listing 2-29 assume the Django messages framework prerequisites are declared in `settings.py` – as described in the sidebar at the beginning of this section.

Because you can end up distributing a Django project to a third party and have no control over the final deployment `settings.py` file, the Django messages framework offers the ability to silently ignore message definitions in case the necessary prerequisites aren't declared in `settings.py`. To silently ignore message definitions if prerequisites aren't declared, you can add the `fail_silently=True` attribute to either technique that adds messages, as illustrated in Listing 2-32.

Listing 2-32. **Use of the fail_silently=True attribute to ignore errors in case Django messages framework not installed**

```
from django.contrib import messages

# Generic add_message method, with fail_silently=True
messages.add_message(request, messages.INFO, 'All items on this page hav

# Shortcut level method, with fail_silently=True
messages.info(request, 'All items on this page have free shipping.',fail
```

Now that you know how to add messages and the important aspects to keep in mind when adding messages, let's take a look at how to access messages.

ACCESS FLASH MESSAGES

The most common place you'll access Django flash messages is in Django templates to display to end users. As a shortcut and thanks to the context processor `django.contrib.messages.context_processors.messages` Django flash messages are available on all templates through the `messages` variable. But before we get to an actual template sample, let's take a quick look at the structure of Django flash messages.

When you add a Django flash message with one of the techniques described in the previous section, Django creates an instance of the `storage.base.Message` class. Table 2-9 describes the structure of the `storage.base.Message` class.

Table 2-9. **Django storage.base.Message structure**

Attribute	Description	Example
message	The actual text of the message.	All items on this page have free shipping.

Attribute	Description	Example
level	An integer describing the type of the message (see Value column in Table 2-8).	20
tags	A string combining all the message tags (extra_tags and level_tag) separated by spaces.	info
extra_tags	A string containing custom tags for this message, separated by spaces.	Empty, by default.
level_tag	The string representation of the level.	info

As you can see in Table 2-9, there are several attributes that you can leverage to display in Django templates. Listing 2-33 shows the boilerplate template code you can use to display all flash messages set in a request.

Listing 2-33. **Boilerplate code to use in Django template to display Django flash messages**

```
{% if messages %}
<ul class="messages">
    {% for msg in messages %}
    <li>
        <div class="alert alert-{{msg.level_tag}}" role="alert">
        {{msg.message}}
        </div>
    </li>
    {% endfor %}
</ul>
{% endif %}
```

Listing 2-33 starts by checking if the `messages` variable exists - which contains all flash messages - if it does, then an HTML list is started with `<ul>`. Next, a loop is made over all the elements in `messages`, where each of these elements corresponds to a `storage.base.Message` instance. For each of these elements, a list and section tag - `<li>` and `<div>` - are created to output the `level_tag` attribute as a CSS class and the message attribute as the `<div>` content.

You can modify the boilerplate code in Listing 2-33 as you see necessary, for example, to include conditionals and output certain message levels or leverage some of the other `storage.base.Message` attributes, among other things.

Note

The HTML code in Listing 2-33 uses the CSS class class="alert alert-{{msg.level_tag}}" that gets rendered into class="alert alert-info" or class="alert alert-success", depending on the level_tag attribute.These CSS classes are part of the CSS bootstrap framework. In this manner, you can quickly format flash messages to look like those presented in Figure 2-2.

Although you'll commonly access Django flash messages in Django templates, this doesn't mean you can't access them elsewhere, such as view methods. You can also gain access to Django flash messages in a request through the `get_messages()` method of the `django.contrib.messages` package. Listing 2-34 illustrates a code snippet with the use of the `get_messages()` method.

Listing 2-34. **Use of get_messages() method to access Django flash messages**

```
from django.contrib import messages

the_req_messages = messages.get_messages(request)
for msg in the_req_messages:
    do_something_with_the_flash_message(msg)
```

In Listing 2-34 the `get_messages()` method receives the `request` as input and assigns the result to `the_req_messages` variable. Next, a loop is made over all the elements in `the_req_messages`, where each of these elements corresponds to a `storage.base.Message` instance. For each of these elements, a call is made to the method `do_something_with_the_flash_message` to do something with each flash message.

An important aspect to understand when accessing Django flash messages is the duration of the messages themselves. Django flash messages are marked to be cleared when an iteration occurs on the main messages instance and cleared when the response is processed.

For access in Django templates, this means that if you fail to make an iteration in a Django template like the one in Listing 2-33 and flash messages are in the request, it can lead to stale or phantom messages appearing elsewhere until an iteration is made and a response is processed. For access in Django view methods (i.e., using `get_messages()`), this has no impact because even though you may make an iteration over the main messages instance - therefore, marking messages to be cleared - a response is not processed in a Django view method, so messages are never cleared, just marked to be cleared.

Class-Based Views

In Chapter 1 and at the start of this chapter – in Listing 2-1 – you saw how to define a Django url and make it operate with a Django template without the need of a view method. This was possible due to the `django.views.generic.TemplateView` class, which is called a class-based view .

Unlike Django view methods backed by standard Python methods that use a Django `HttpRequest` input parameter and output a Django `HttpResponse`, class-based views offer their functionality through full-fledged Python classes. This, in turn, allows Django views to operate with object-oriented programming (OOP) principles (e.g., encapsulation, polymorphism, and inheritance) leading to greater reusability and shorter implementation times.

Although Django class-based views represent a more powerful approach to create Django views, they are simply an alternative to the view methods you've used up to this point. If you want to quickly execute business logic on Django requests you can keep using view methods, but for more demanding view requirements (e.g., form processing, boilerplate model queries) class-based views can save you considerable time.

BUILT-IN CLASS-BASED VIEWS

The functionality provided by the `django.views.generic.TemplateView` class-based view is really a time saver. While it would have been possible to configure a url to execute on an empty view method and then send control to a template, the `TemplateView` class allows this process to be done in one line.

In addition to the `TemplateView` class-based view, Django offers many other built-in class-based views to shorten the creation process for common Django view operations using OOP-like principles. Table 2-10 illustrates Django's built-in classes for views.

Table 2-10. **Built-in classes for views**

Class	Description
django.views.generic.View	Parent class of all class-based views, providing core functionality.
django.views.generic.TemplateView	Allows a url to return the contents of a template, without the need of a view.
django.views.generic.RedirectView	Allows a url to perform a redirect, without the need of a view.

Class	Description
django.views.generic.ArchiveIndexView django.views.generic.YearArchiveView django.views.generic.MonthArchiveView django.views.generic.WeekArchiveView django.views.generic.DayArchiveView django.views.generic.TodayArchiveView django.views.generic.DateDetailView	Allows a view to return date-based object results, without the need to explicitly perform Django model queries.
django.views.generic.CreateView django.views.generic.DetailView django.views.generic.UpdateView django.views.generic.DeleteView django.views.generic.ListView django.views.generic.FormView	Allows a view to execute Create-Read-Update-Delete (CRUD) operations , without the need to explicitly perform Django model queries.

In the upcoming and final section of this chapter, I'll explain the classes in the top half of Table 2-10 so you can gain a better understanding of the structure and execution process of Django class-based views. The class-based views in the bottom half of Table 2-10 that involve Django models are described in a separate chapter on Django models.

CLASS-BASED VIEW STRUCTURE AND EXECUTION

To create a class-based view you need to create a class that inherits from one of the classes in Table 2-10. Listing 2-35 shows a class-based view with this inheritance technique, as well as the corresponding url definition to execute a class-based view.

Listing 2-35. **Class-based view inherited from TemplateView with url definition**

```
# views.py
from django.views.generic import TemplateView

class AboutIndex(TemplateView):
      template_name = 'index.html'

      def get_context_data(self, **kwargs):
         # **kwargs contains keyword context initialization values (if a
         # Call base implementation to get a context
         context = super(AboutIndex, self).get_context_data(**kwargs)
         # Add context data to pass to template
         context['aboutdata'] = 'Custom data'
         return context
```

```
#urls.py
from coffeehouse.about.views import AboutIndex

urlpatterns = [
    url(r'^about/index/',AboutIndex.as_view(),{'onsale':True}),
]
```

I chose to create a view that inherits from `TemplateView` first because of its simplicity and because you already know the purpose of this class. The example in Listing 2-35 and the first example in this chapter from Listing 2-1 produce nearly identical outcomes.

The difference is, Listing 2-1 declares a `TemplateView` class instance directly as part of the url (e.g., `TemplateView.as_view(template_name='index.html'))`), where as Listing 2-35 declares an instance of a `TemplateView` subclass named `AboutIndex`. Comparing the two approaches, you can get the initial feel for the OOP behavior of class-based views.

The first part in Listing 2-35 declares the `AboutIndex` class-based view, which inherits its behavior from the `TemplateView` class. Notice the class declares the `template_name` attribute and the `get_context_data()` method.

The `template_name` value in the `AboutIndex` class acts as a default template for the class-based view. But in OOP fashion, this same value can be overridden by providing a value at instance creation (e.g., `AboutIndex.as_view(template_name='other.html')` to use the `other.html` template).

The `get_context_data` method in the `AboutIndex` class allows you to add context data to the class-view template. Notice the signature of the `get_context_data` method uses `**kwargs` to gain access to context initialization values (e.g., declared in the url or parent class-views) and invokes a parent's class `get_context_data` method using the Python `super()` method per standard OOP Python practice. Next, the `get_context_data` method adds the additional context data with the `aboutdata` key and returns the modified `context` reference.

In the second part of Listing 2-35, you can see how the `AboutIndex` class-based view is first imported into a `urls.py` file and then hooked up to a url definition. Notice how the class-based view is declared on the `url` definition using the `as_view()` method. In addition, notice how the url definition declares the url extra option `{'onsale':True}` that gets passed as context data to the class-based view (i.e., in the `**kwargs` of the `get_context_data` method).

Tip

All class-based views use the as_view() method to integrate into url definitions.

Now that you have a basic understanding of Django class-based views, Listing 2-36 shows another class-based view with different implementation details.

Listing 2-36. **Class-based view inherited from View with multiple HTTP handling**

```
# views.py
from django.views.generic import View
from django.http import HttpResponse
from django.shortcuts import render

class ContactPage(View):
    mytemplate = 'contact.html'
    unsupported = 'Unsupported operation'

    def get(self, request):
        return render(request, self.mytemplate)

    def post(self, request):
        return HttpResponse(self.unsupported)

#urls.py
from coffeehouse.contact.views import ContactPage

urlpatterns = [
    url(r'^contact/$',ContactPage.as_view()),
]
```

The first difference in Listing 2-36 is the class-based view inherits its behavior from the general purpose `django.views.generic.View` class. As outlined in Table 2-10, the `View` class provides the core functionality for all class-based views. So in fact, the `TemplateView` class used in Listing 2-35 is a subclass of `View`, meaning class-based views that use `TemplateView` have access to the same functionalities of class-based views that use `View`.

The reason you would chose one class over another to implement class-based views is rooted in OOP polymorphism principles. For example, in OOP you can have a class hierarchy Drink→ Coffee → Latte, where a Drink class offers generic functionalities available to Drink, Coffee, and Latte instances; a Coffee class offers more specific functionalities applicable to Coffee and Latter instances; and a Latte class offers the most specific functionalities applicable to only Latte instances.

Therefore if you know beforehand you need a class-based view to relinquish control to a template without applying elaborate business logic or custom request and response handling, the `TemplateView` class offers the quickest path to a solution vs. the more generic `View` class. Expanding on this same principle, once you start working with Django models and views, you'll come to realize some of the more specialized class-based views in Table 2-10 also offer quicker solutions than creating a class-based view that inherits from the general purpose `View` class. Now that you know the reason why you would chose a `View` class-based view over a more specialized class, let's break down the functionality in Listing 2-36.

Notice the class-based view `ContactPage` declares two attributes: `mytemplate` and `unsupported`. These are generic class attributes and I used the `mytemplate` name to illustrate there's no relation to the `template_name` attribute used in Listing 2-35 and `TemplateView` class-based views. Class-based views derived from a `TemplateView` *expect* a `template_name` value and automatically use this template to generate a response. However, class-based views derived from a `View` class don't expect a specific template, but instead expect you to implement how to generate a response, which is where the `get` and `post` methods in Listing 2-36 come into play.

The `get` method is used to handle HTTP GET requests on the view, while the `post` method is used to HTTP POST requests on the view. This offers a much more modular approach to handle different HTTP operations vs. standard view methods that require explicitly inspecting a request and creating conditionals to handle different HTTP operations. For the moment, don't worry about HTTP GET and HTTP POST view handling; this is explored in greater detail in Django forms where the topic is of greater relevance.

Next, notice both the `get` and `post` methods declare a `request` input, which represents a Django `HttpRequest` instance just like standard view methods. In both cases, the methods immediately return a response, but it's possible to inspect a request value or execute any business logic before generating a response, just like it can be done in standard view methods.

The `get` method generates a response with the `django.shortcuts.render` method and the `post` method generates a response with the `HttpResponse` class, both of which are the same techniques used to generate responses in standard view methods . The only minor difference in Listing 2-36 is both the `render` method and `HttpResponse` class use instance attributes (e.g., `self.mytemplate`, `self.unsupported`) to generate the response, but other than this, you're free to return a Django `HttpResponse` with any of the variations already explained in this chapter (e.g., Listing 2-22 response alternatives, Table 2-5 shortcut responses).

Finally, the last part in Listing 2-36 shows how the `ContactPage` class-based view is imported into a `urls.py` file and later hooked up to a url using the `as_view()` method.

To close out the discussion on class-based views and this chapter, we come to the `django.views.generic.RedirectView` class. Similar to the `TemplateView` class-based view that allows you to quickly generate a response without a view method, the `RedirectView` class-based view allows you to quickly generate an HTTP redirect – like the ones described in Table 2-4 – without the need of a view method.

The `RedirectView` class supports four attributes described in the following list:

- `permanent`.- Defaults to `False` to perform a non-permanent redirect supported by the `HttpResponseRedirect` class described in Table 2-4. If set to `True`, a permanent redirect is made with the `HttpResponsePermanentRedirect` class described in Table 2-4.
- `url`.- Defaults to `None`. Defines a url value to perform the redirect.
- `pattern_name`.- Defaults to `None`. Defines a url name to generate a redirect url via the `reverse` method. Note the `reverse` method is explained in the url naming and namespace section earlier in this chapter.
- `query_string`.- Defaults to `False` to append a query string to a redirect url. If provided, the `query_string` value to the redirect url.

And with this we conclude our exploration into Django views and urls. In the next two chapters, you'll learn about Django templates and Jinja templates.

Footnotes

1 http://www.apress.com/la/book/9781590594414 (http://www.apress.com/la/book/9781590594414)

2 https://docs.python.org/3/howto/regex.html#non-capturing-and-named-groups

3 https://docs.djangoproject.com/en/1.11/_modules/django/http/request/#HttpRequest

4 https://docs.djangoproject.com/en/1.11/_modules/django/http/request/#QueryDict

5 https://docs.djangoproject.com/en/1.11/ref/request-response/#django.http.HttpResponse

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
1. Introduction to the Django Framework

NEXT
3. Django Templates