URL dispatcher

A clean, elegant URL scheme is an important detail in a high-quality web application. Django lets you design URLs however you want, with no framework limitations.

See Cool URIs don't change, by World Wide Web creator Tim Berners-Lee, for excellent arguments on why URLs should be clean and usable.

Overview

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically.

Django also provides a way to translate URLs according to the active language. See the internationalization documentation for more information.

How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the `ROOT_URLCONF` setting, but if the incoming `HttpRequest` object has a `urlconf` attribute (set by middleware), its value will be used in place of the `ROOT_URLCONF` setting.
2. Django loads that Python module and looks for the variable `urlpatterns`. This should be a sequence of `django.urls.path()` and/or `django.urls.re_path()` instances.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL, matching against `path_info`.
4. Once one of the URL patterns matches, Django imports and calls the given view, which is a Python function (or a class-based view). The view gets passed the following arguments:
   - An instance of `HttpRequest`.
   - If the matched URL pattern contained no named groups, then the matches from the regular expression are provided as positional arguments.
   - The keyword arguments are made up of any named parts matched by the path expression that are provided, overridden by any arguments specified in the optional `kwargs` argument to `django.urls.path()` or `django.urls.re_path()`.
5. If no URL pattern matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. See Error handling below.

Example

Here’s a sample URLconf:

```
from django.urls import path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<int:year>/", views.year_archive),
    path("articles/<int:year>/<int:month>/", views.month_archive),
    path("articles/<int:year>/<int:month>/<slug:slug>/", views.article_detail),
]
```

Notes:

- To capture a value from the URL, use angle brackets.
- Captured values can optionally include a converter type. For example, use `<int:name>` to capture an integer parameter. If a converter isn’t included, any string, excluding a `/` character, is matched.
- There’s no need to add a leading slash, because every URL has that. For example, it’s `articles`, not `/articles`.

Example requests:

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `views.month_archive(request, year=2005, month=3)`.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this. Here, Django would call the function `views.special_case_2003(request)`
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/building-a-django-site/` would match the final pattern. Django would call the function `views.article_detail(request, year=2003, month=3, slug="building-a-django-site")`.

Path converters

The following path converters are available by default:

- `str` - Matches any non-empty string, excluding the path separator, `'/'`. This is the default if a converter isn’t included in the expression.
- `int` - Matches zero or any positive integer. Returns an `int`.
- `slug` - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, `building-your-1st-django-site`.
- `uuid` - Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, `075194d3-6885-417e-a8a8-6c931e272f00`. Returns a `UUID` instance.

- `path` - Matches any non-empty string, including the path separator, `'/'`. This allows you to match against a complete URL path rather than a segment of a URL path as with `str`.

Registering custom path converters

For more complex matching requirements, you can define your own path converters.

A converter is a class that includes the following:

- A `regex` class attribute, as a string.
- A `to_python(self, value)` method, which handles converting the matched string into the type that should be passed to the view function. It should raise `ValueError` if it can't convert the given value. A `ValueError` is interpreted as no match and as a consequence a 404 response is sent to the user unless another URL pattern matches.
- A `to_url(self, value)` method, which handles converting the Python type into a string to be used in the URL. It should raise `ValueError` if it can't convert the given value. A `ValueError` is interpreted as no match and as a consequence `reverse()` will raise `NoReverseMatch` unless another URL pattern matches.

For example:

```
class FourDigitYearConverter:
    regex = "[0-9]{4}"

    def to_python(self, value):
        return int(value)

    def to_url(self, value):
        return "%04d" % value
```

Register custom converter classes in your URLconf using `register_converter()`:

```
from django.urls import path, register_converter

from . import converters, views

register_converter(converters.FourDigitYearConverter, "yyyy")

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<yyyy:year>/", views.year_archive),
    ...,
]
```

Deprecated since version 5.1:

Overriding existing converters with `django.urls.register_converter()` is deprecated.

Using regular expressions

If the paths and converters syntax isn't sufficient for defining your URL patterns, you can also use regular expressions. To do so, use `re_path()` instead of `path()`.

In Python regular expressions, the syntax for named regular expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

Here's the example URLconf from earlier, rewritten using regular expressions:

```
from django.urls import path, re_path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    re_path(r"^articles/(?P<year>[0-9]{4})/$", views.year_archive),
    re_path(r"^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$", views.month_archive),
    re_path(
        r"^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$",
        views.article_detail,
    ),
]
```

This accomplishes roughly the same thing as the previous example, except:

- The exact URLs that will match are slightly more constrained. For example, the year 10000 will no longer match since the year integers are constrained to be exactly four digits long.
- Each captured argument is sent to the view as a string, regardless of what sort of match the regular expression makes.

When switching from using `path()` to `re_path()` or vice versa, it's particularly important to be aware that the type of the view arguments may change, and so you may need to adapt your views.

Using unnamed regular expression groups

As well as the named group syntax, e.g. `(?P<year>[0-9]{4})`, you can also use the shorter unnamed group, e.g. `([0-9]{4})`.

This usage isn't particularly recommended as it makes it easier to accidentally introduce errors between the intended meaning of a match and the arguments of the view.

In either case, using only one style within a given regex is recommended. When both styles are mixed, any unnamed groups are ignored and only named groups are passed to the view function.

Nested arguments

Regular expressions allow nested arguments, and Django will resolve them and pass them to the view. When reversing, Django will try to fill in all outer captured arguments, ignoring any nested captured arguments. Consider the following URL patterns which optionally take a page argument:

```
from django.urls import re_path

urlpatterns = [
    re_path(r"^blog/(page-([0-9]+)/)?$", blog_articles),  # bad
    re_path(r"^comments/(?:page-(?P<page_number>[0-9]+)/)?$", comments),  # good
]
```

Both patterns use nested arguments and will resolve: for example, `blog/page-2/` will result in a match to `blog_articles` with two positional arguments: `page-2/` and `2`. The second pattern for `comments` will match `comments/page-2/` with keyword argument `page_number` set to 2. The outer argument in this case is a non-capturing argument `(?:...)`.

The `blog_articles` view needs the outermost captured argument to be reversed, `page-2/` or no arguments in this case, while `comments` can be reversed with either no arguments or a value for `page_number`.

Nested captured arguments create a strong coupling between the view arguments and the URL as illustrated by `blog_articles`: the view receives part of the URL (`page-2/`) instead of only the value the view is interested in. This coupling is even more pronounced when reversing, since to reverse the view we need to pass the piece of URL instead of the page number.

As a rule of thumb, only capture the values the view needs to work with and use non-capturing arguments when the regular expression needs an argument but the view ignores it.

What the URLconf searches against

The URLconf searches against the requested URL, as a normal Python string. This does not include GET or POST parameters, or the domain name.

For example, in a request to `https://www.example.com/myapp/`, the URLconf will look for `myapp/`.

In a request to `https://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

The URLconf doesn’t look at the request method. In other words, all request methods – `POST`, `GET`, `HEAD`, etc. – will be routed to the same function for the same URL.

Specifying defaults for view arguments

A convenient trick is to specify default parameters for your views’ arguments. Here’s an example URLconf and view:

```
# URLconf
from django.urls import path

from . import views

urlpatterns = [
    path("blog/", views.page),
    path("blog/page<int:num>/", views.page),
]


# View (in blog/views.py)
def page(request, num=1):
    # Output the appropriate page of blog entries, according to num.
    ...
```

In the above example, both URL patterns point to the same view – `views.page` – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `1`. If the second pattern matches, `page()` will use whatever `num` value was captured.

Performance

Django processes regular expressions in the `urlpatterns` list which is compiled the first time it's accessed. Subsequent requests use the cached configuration via the URL resolver.

Syntax of the `urlpatterns` variable

`urlpatterns` should be a sequence of `path()` and/or `re_path()` instances.

Error handling

When Django can't find a match for the requested URL, or when an exception is raised, Django invokes an error-handling view.

The views to use for these cases are specified by four variables. Their default values should suffice for most projects, but further customization is possible by overriding their default values.

See the documentation on customizing error views for the full details.

Such values can be set in your root URLconf. Setting these variables in any other URLconf will have no effect.

Values must be callables, or strings representing the full Python import path to the view that should be called to handle the error condition at hand.

The variables are:

- `handler400` – See `django.conf.urls.handler400`.
- `handler403` – See `django.conf.urls.handler403`.
- `handler404` – See `django.conf.urls.handler404`.
- `handler500` – See `django.conf.urls.handler500`.

Including other URLconfs

At any point, your `urlpatterns` can “include” other URLconf modules. This essentially “roots” a set of URLs below other ones.

For example, here’s an excerpt of the URLconf for the Django website itself. It includes a number of other URLconfs:

```
from django.urls import include, path

urlpatterns = [
    # ... snip ...
    path("community/", include("aggregator.urls")),
    path("contact/", include("contact.urls")),
    # ... snip ...
]
```

Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Another possibility is to include additional URL patterns by using a list of `path()` instances. For example, consider this URLconf:

```
from django.urls import include, path

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    path("reports/", credit_views.report),
    path("reports/<int:id>/", credit_views.report),
    path("charge/", credit_views.charge),
]

urlpatterns = [
    path("", main_views.homepage),
    path("help/", include("apps.help.urls")),
    path("credit/", include(extra_patterns)),
]
```

In this example, the `/credit/reports/` URL will be handled by the `credit_views.report()` Django view.

This can be used to remove redundancy from URLconfs where a single pattern prefix is used repeatedly. For example, consider this URLconf:

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

urlpatterns = [
    path("<page_slug>-<page_id>/history/", views.history),
    path("<page_slug>-<page_id>/edit/", views.edit),
    path("<page_slug>-<page_id>/discuss/", views.discuss),
    path("<page_slug>-<page_id>/permissions/", views.permissions),
]
```

We can improve this by stating the common path prefix only once and grouping the suffixes that differ:

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

urlpatterns = [
    path(
        "<page_slug>-<page_id>/",
        include(
            [
                path("history/", views.history),
                path("edit/", views.edit),
                path("discuss/", views.discuss),
                path("permissions/", views.permissions),
            ]
        ),
    ),
]
```

Captured parameters

An included URLconf receives any captured parameters from parent URLconfs, so the following example is valid:

```
# In settings/urls/main.py
from django.urls import include, path

urlpatterns = [
    path("<username>/blog/", include("foo.urls.blog")),
]

# In foo/urls/blog.py
from django.urls import path
from . import views

urlpatterns = [
    path("", views.blog.index),
    path("archive/", views.blog.archive),
]
```

In the above example, the captured `"username"` variable is passed to the included URLconf, as expected.

Passing extra options to view functions

URLconfs have a hook that lets you pass extra arguments to your view functions, as a Python dictionary.

The `path()` function can take an optional third argument which should be a dictionary of extra keyword arguments to pass to the view function.

For example:

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

urlpatterns = [
    path("blog/<int:year>/", views.year_archive, {"foo": "bar"}),
]
```

In this example, for a request to `/blog/2005/`, Django will call `views.year_archive(request, year=2005, foo='bar')`.

This technique is used in the syndication framework to pass metadata and options to views.

Dealing with conflicts: It's possible to have a URL pattern which captures named keyword arguments, and also passes arguments with the same names in its dictionary of extra arguments. When this happens, the arguments in the dictionary will be used instead of the arguments captured in the URL.

Passing extra options to `include()`

Similarly, you can pass extra options to `include()` and each line in the included URLconf will be passed the extra options.

For example, these two URLconf sets are functionally identical:

Set one:

```
# main.py
from django.urls import include, path

urlpatterns = [
    path("blog/", include("inner"), {"blog_id": 3}),
]

# inner.py
from django.urls import path
from mysite import views

urlpatterns = [
    path("archive/", views.archive),
    path("about/", views.about),
]
```

Set two:

```
# main.py
from django.urls import include, path
from mysite import views

urlpatterns = [
    path("blog/", include("inner")),
]

# inner.py
from django.urls import path

urlpatterns = [
    path("archive/", views.archive, {"blog_id": 3}),
    path("about/", views.about, {"blog_id": 3}),
]
```

Note that extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is only useful if you're certain that every view in the included URLconf accepts the extra options you're passing.

Reverse resolution of URLs

A common need when working on a Django project is the possibility to obtain URLs in their final forms either for embedding in generated content (views and assets URLs, URLs shown to the user, etc.) or for handling of the navigation flow on the server side (redirections, etc.)

It is strongly desirable to avoid hard-coding these URLs (a laborious, non-scalable and error-prone strategy). Equally dangerous is devising ad-hoc mechanisms to generate URLs that are parallel to the design described by the URLconf, which can result in the production of URLs that become stale over time.

In other words, what's needed is a DRY mechanism. Among other advantages it would allow evolution of the URL design without having to go over all the project source code to search and replace outdated URLs.

The primary piece of information we have available to get a URL is an identification (e.g. the name) of the view in charge of handling it. Other pieces of information that necessarily must participate in the lookup of the right URL are the types (positional, keyword) and values of the view arguments.

Django provides a solution such that the URL mapper is the only repository of the URL design. You feed it with your URLconf and then it can be used in both directions:

- Starting with a URL requested by the user/browser, it calls the right Django view providing any arguments it might need with their values as extracted from the URL.
- Starting with the identification of the corresponding Django view plus the values of arguments that would be passed to it, obtain the associated URL.

The first one is the usage we've been discussing in the previous sections. The second one is what is known as *reverse resolution of URLs*, *reverse URL matching*, *reverse URL lookup*, or simply *URL reversing*.

Django provides tools for performing URL reversing that match the different layers where URLs are needed:

- In templates: Using the `url` template tag.
- In Python code: Using the `reverse()` function.
- In higher level code related to handling of URLs of Django model instances: The `get_absolute_url()` method.

Examples

Consider again this URLconf entry:

```
from django.urls import path

from . import views

urlpatterns = [
    # ...
    path("articles/<int:year>/", views.year_archive, name="news-year-archive"),
    # ...
]
```

According to this design, the URL for the archive corresponding to year *nnnn* is `/articles/<nnnn>/`.

You can obtain these in template code by using:

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>
{# Or with the year in a template context variable: #}
<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

Or in Python code:

```
from django.http import HttpResponseRedirect
from django.urls import reverse


def redirect_to_year(request):
    # ...
    year = 2006
    # ...
    return HttpResponseRedirect(reverse("news-year-archive", args=(year,)))
```

If, for some reason, it was decided that the URLs where content for yearly article archives are published at should be changed then you would only need to change the entry in the URLconf.

In some scenarios where views are of a generic nature, a many-to-one relationship might exist between URLs and views. For these cases the view name isn't a good enough identifier for it when comes the time of reversing URLs. Read the next section to know about the solution Django provides for this.

Naming URL patterns

In order to perform URL reversing, you'll need to use **named URL patterns** as done in the examples above. The string used for the URL name can contain any characters you like. You are not restricted to valid Python names.

When naming URL patterns, choose names that are unlikely to clash with other applications' choice of names. If you call your URL pattern `comment` and another application does the same thing, the URL that `reverse()` finds depends on whichever pattern is last in your project's `urlpatterns` list.

Putting a prefix on your URL names, perhaps derived from the application name (such as `myapp-comment` instead of `comment`), decreases the chance of collision.

You can deliberately choose the *same URL name* as another application if you want to override a view. For example, a common use case is to override the `LoginView`. Parts of Django and most third-party apps assume that this view has a URL pattern with the name `login`. If you have a custom login view and give its URL the name `login`, `reverse()` will find your custom view as long as it's in `urlpatterns` after `django.contrib.auth.urls` is included (if that's included at all).

You may also use the same name for multiple URL patterns if they differ in their arguments. In addition to the URL name, `reverse()` matches the number of arguments and the names of the keyword arguments. Path converters can also raise `ValueError` to indicate no match, see Registering custom path converters for details.

URL namespaces

Introduction

URL namespaces allow you to uniquely reverse named URL patterns even if different applications use the same URL names. It's a good practice for third-party apps to always use namespaced URLs (as we did in the tutorial). Similarly, it also allows you to reverse URLs if multiple instances of an application are deployed. In other words, since multiple instances of a single application will share named URLs, namespaces provide a way to tell these named URLs apart.

Django applications that make proper use of URL namespacing can be deployed more than once for a particular site. For example `django.contrib.admin` has an `AdminSite` class which allows you to deploy more than one instance of the admin. In a later example, we'll discuss the idea of deploying the polls application from the tutorial in two different locations so we can serve the same functionality to two different audiences (authors and publishers).

A URL namespace comes in two parts, both of which are strings:

`application namespace`

This describes the name of the application that is being deployed. Every instance of a single application will have the same application namespace. For example, Django's admin application has the somewhat predictable application namespace of `'admin'` .

`instance namespace`

This identifies a specific instance of an application. Instance namespaces should be unique across your entire project. However, an instance namespace can be the same as the application namespace. This is used to specify a default instance of an application. For example, the default Django admin instance has an instance namespace of `'admin'` .

Namespaced URLs are specified using the `':'` operator. For example, the main index page of the admin application is referenced using `'admin:index'` . This indicates a namespace of `'admin'` , and a named URL of `'index'` .

Namespaces can also be nested. The named URL `'sports:polls:index'` would look for a pattern named `'index'` in the namespace `'polls'` that is itself defined within the top-level namespace `'sports'` .

Reversing namespaced URLs

When given a namespaced URL (e.g. `'polls:index'`) to resolve, Django splits the fully qualified name into parts and then tries the following lookup:

1. First, Django looks for a matching application namespace (in this example, `'polls'`). This will yield a list of instances of that application.
2. If there is a current application defined, Django finds and returns the URL resolver for that instance. The current application can be specified with the `current_app` argument to the `reverse()` function.

   The `url` template tag uses the namespace of the currently resolved view as the current application in a `RequestContext`. You can override this default by setting the current application on the `request.current_app` attribute.
3. If there is no current application, Django looks for a default application instance. The default application instance is the instance that has an instance namespace matching the application namespace (in this example, an instance of `polls` called `'polls'`).
4. If there is no default application instance, Django will pick the last deployed instance of the application, whatever its instance name may be.
5. If the provided namespace doesn't match an application namespace in step 1, Django will attempt a direct lookup of the namespace as an instance namespace.

If there are nested namespaces, these steps are repeated for each part of the namespace until only the view name is unresolved. The view name will then be resolved into a URL in the namespace that has been found.

Example

To show this resolution strategy in action, consider an example of two instances of the `polls` application from the tutorial: one called `'author-polls'` and one called `'publisher-polls'` . Assume we have enhanced that application so that it takes the instance namespace into consideration when creating and displaying polls.

`urls.py`

```
from django.urls import include, path

urlpatterns = [
    path("author-polls/", include("polls.urls", namespace="author-polls")),
    path("publisher-polls/", include("polls.urls", namespace="publisher-polls")),
]
```

`polls/urls.py`

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.IndexView.as_view(), name="index"),
    path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ...,
]
```

Using this setup, the following lookups are possible:

- If one of the instances is current - say, if we were rendering the detail page in the instance `'author-polls'` - `'polls:index'` will resolve to the index page of the `'author-polls'` instance; i.e. both of the following will result in `"/author-polls/"`.

  In the method of a class-based view:

  ```
  reverse("polls:index", current_app=self.request.resolver_match.namespace)
  ```

  and in the template:

  ```
  {% url 'polls:index' %}
  ```

- If there is no current instance - say, if we were rendering a page somewhere else on the site - `'polls:index'` will resolve to the last registered instance of `polls`. Since there is no default instance (instance namespace of `'polls'`), the last instance of `polls` that is registered will be used. This would be `'publisher-polls'` since it's declared last in the `urlpatterns`.
- `'author-polls:index'` will always resolve to the index page of the instance `'author-polls'` (and likewise for `'publisher-polls'`).

If there were also a default instance - i.e., an instance named `'polls'` - the only change from above would be in the case where there is no current instance (the second item in the list above). In this case `'polls:index'` would resolve to the index page of the default instance instead of the instance declared last in `urlpatterns`.

URL namespaces and included URLconfs

Application namespaces of included URLconfs can be specified in two ways.

Firstly, you can set an `app_name` attribute in the included URLconf module, at the same level as the `urlpatterns` attribute. You have to pass the actual module, or a string reference to the module, to `include()`, not the list of `urlpatterns` itself.

`polls/urls.py`

```
from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.IndexView.as_view(), name="index"),
    path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ...,
]
```

`urls.py`

```
from django.urls import include, path

urlpatterns = [
    path("polls/", include("polls.urls")),
]
```

The URLs defined in `polls.urls` will have an application namespace `polls`.

Secondly, you can include an object that contains embedded namespace data. If you `include()` a list of `path()` or `re_path()` instances, the URLs contained in that object will be added to the global namespace. However, you can also `include()` a 2-tuple containing:

```
(<list of path()/re_path() instances>, <application namespace>)
```

For example:

```
from django.urls import include, path

from . import views

polls_patterns = (
    [
        path("", views.IndexView.as_view(), name="index"),
        path("<int:pk>/", views.DetailView.as_view(), name="detail"),
    ],
    "polls",
)

urlpatterns = [
    path("polls/", include(polls_patterns)),
]
```

This will include the nominated URL patterns into the given application namespace.

The instance namespace can be specified using the `namespace` argument to `include()`. If the instance namespace is not specified, it will default to the included URLconf's application namespace. This means it will also be the default instance for that namespace.