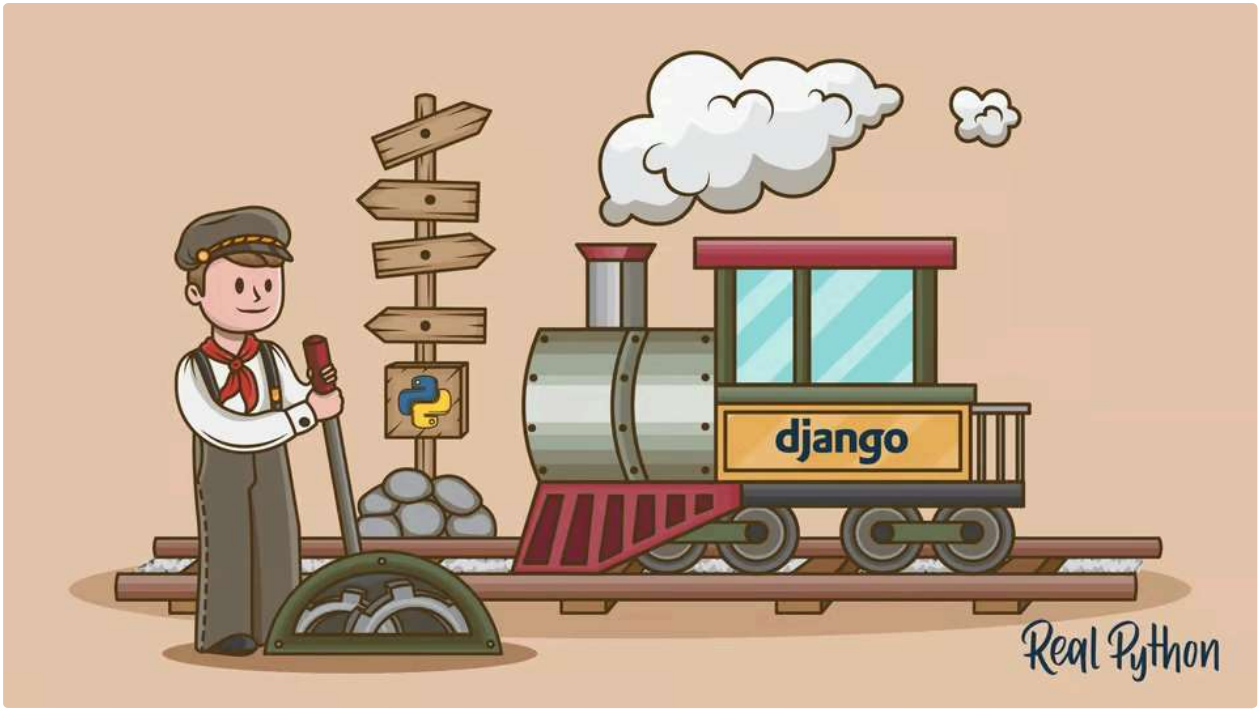

The Ultimate Guide to Django Redirects

by Daniel Hepper Aug 15, 2018 5 Comments django intermediate web-dev

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Django Redirects**

When you build a Python web application with the Django framework, you'll at some point have to redirect the user from one URL to another.

In this guide, you'll learn everything you need to know about HTTP redirects and how to deal with them in Django. At the end of this tutorial, you'll:

- Be able to redirect a user from one URL to another URL
- Know the difference between temporary and permanent redirects
- Avoid common pitfalls when working with redirects

This tutorial assumes that you're familiar with the basic building blocks of a Django application, like views and URL patterns.

Django Redirects: A Super Simple Example

In Django, you redirect the user to another URL by returning an instance of `HttpResponseRedirect` or `HttpResponsePermanentRedirect` from your view. The simplest way to do this is to use the function `redirect()` from the module `django.shortcuts`. Here's an example:

Python

```
# views.py
from django.shortcuts import redirect

def redirect_view(request):
    response = redirect('/redirect-success/')
    return response
```

Just call `redirect()` with a URL in your view. It will return a `HttpResponseRedirect` class, which you then return from your view.

A view returning a redirect has to be added to your `urls.py`, like any other view:

Python

```
# urls.py
from django.urls import path

from .views import redirect_view

urlpatterns = [
    path('/redirect/', redirect_view)
    # ... more URL patterns here
]
```

Assuming this is the main `urls.py` of your Django project, the URL `/redirect/` now redirects to `/redirect-success/`.

To avoid hard-coding the URL, you can call `redirect()` with the name of a view or URL pattern or a model to avoid hard-coding the redirect URL. You can also create a permanent redirect by passing the keyword argument `permanent=True`.

This article could end here, but then it could hardly be called "The Ultimate Guide to Django Redirects." We will take a closer look at the `redirect()` function in a minute and also get into the nitty-gritty details of HTTP status codes and different `HttpRedirectResponse` classes, but let's take a step back and start with a fundamental question.

Why Redirect

You might wonder why you'd ever want to redirect a user to a different URL in the first place. To get an idea where redirects make sense, have a look at how Django itself incorporates redirects into features that the framework provides by default:

- When you are not logged-in and request a URL that requires authentication, like the Django admin, Django redirects you to the login page.
- When you log in successfully, Django redirects you to the URL you requested originally.
- When you change your password using the Django admin, you are redirected to a page that indicates that the change was successful.
- When you create an object in the Django admin, Django redirects you to the object list.

What would an alternative implementation without redirects look like? If a user has to log in to view a page, you could simply display a page that says something like "Click here to log in." This would work, but it would be inconvenient for the user.

URL shorteners like http://bit.ly are another example of where redirects come in handy: you type a short URL into the address bar of your browser and are then redirected to a page with a long, unwieldy URL.

> **Note:** If you'd like to build a URL shortener of your own, then check out Build a URL Shortener With FastAPI and Python.

In other cases, redirects are not just a matter of convenience. Redirects are an essential instrument to guide the user through a web application. After performing some kind of operation with side effects, like creating or deleting an object, it's a best practice to redirect to another URL to prevent accidentally performing the operation twice.

One example of this use of redirects is form handling, where a user is redirected to another URL after successfully submitting a form. Here's a code sample that illustrates how you'd typically handle a form:

Python

```python
from django import forms
from django.http import HttpResponseRedirect
from django.shortcuts import redirect, render

def send_message(name, message):
    # Code for actually sending the message goes here

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

def contact_view(request):
    # The request method 'POST' indicates
    # that the form was submitted
    if request.method == 'POST':  # 1
        # Create a form instance with the submitted data
        form = ContactForm(request.POST)  # 2
        # Validate the form
        if form.is_valid():  # 3
            # If the form is valid, perform some kind of
            # operation, for example sending a message
            send_message(
                form.cleaned_data['name'],
                form.cleaned_data['message']
            )
            # After the operation was successful,
            # redirect to some other page
            return redirect('/success/')  # 4
    else:  # 5
        # Create an empty form instance
        form = ContactForm()

    return render(request, 'contact_form.html', {'form': form})
```

The purpose of this view is to display and handle a contact form that allows the user to send a message. Let's follow it step by step:

1. First the view looks at the request method. When the user visits the URL connected to this view, the browser performs a `GET` request.

2. If the view is called with a `POST` request, the `POST` data is used to instantiate a `ContactForm` object.

3. If the form is valid, the form data is passed to `send_message()`. This function is not relevant in this context and therefore not shown here.

4. After sending the message, the view returns a redirect to the URL `/success/`. This is the step we are interested in. For simplicity, the URL is hard-coded here. You'll see later how you can avoid that.

5. If the view receives a `GET` request (or, to be precise, any kind of request that is not a `POST` request), it creates an instance of `ContactForm` and uses `django.shortcuts.render()` to render the `contact_form.html` template.

If the user now hits reload, only the `/success/` URL is reloaded. Without the redirect, reloading the page would re-submit the form and send another message.

Behind the Scenes: How an HTTP Redirect Works

Now you know why redirects make sense, but how do they work? Let's have a quick recap of what happens when you enter a URL in the address bar of your web browser.

A Quick Primer on HTTP

Let's assume you've created a Django application with a "Hello World" view that handles the path `/hello/`. You are running your application with the Django development server, so the complete URL is `http://127.0.0.1:8000/hello/`.

When you enter that URL in your browser, it connects to port `8000` on the server with the IP address `127.0.0.1` and sends an HTTP `GET` request for the path `/hello/`. The server replies with an HTTP response.

HTTP is text-based, so it's relatively easy to look at the back and forth between the client and the server. You can use the command line tool `curl` with the option `--include` to have a look at the complete HTTP response including the headers, like this:

Shell

```
$ curl --include http://127.0.0.1:8000/hello/
HTTP/1.1 200 OK
Date: Sun, 01 Jul 2018 20:32:55 GMT
Server: WSGIServer/0.2 CPython/3.6.3
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 11

Hello World
```

As you can see, an HTTP response starts with a status line that contains a status code and a status message. The status line is followed by an arbitrary number of HTTP headers. An empty line indicates the end of the headers and the start of the response body, which contains the actual data the server wants to send.

HTTP Redirects Status Codes

What does a redirect response look like? Let's assume the path `/redirect/` is handled by `redirect_view()`, shown earlier. If you access `http://127.0.0.1:8000/redirect/` with `curl`, your console looks like this:

Shell

```
$ curl --include http://127.0.0.1:8000/redirect/
HTTP/1.1 302 Found
Date: Sun, 01 Jul 2018 20:35:34 GMT
Server: WSGIServer/0.2 CPython/3.6.3
Content-Type: text/html; charset=utf-8
Location: /redirect-success/
X-Frame-Options: SAMEORIGIN
Content-Length: 0
```

The two responses might look similar, but there are some key differences. The redirect:

- Returns a different status code (`302` versus `200`)
- Contains a `Location` header with a relative URL
- Ends with an empty line because the body of the redirect response is empty

The primary differentiator is the status code. The specification of the HTTP standard says the following:

> The 302 (Found) status code indicates that the target resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client ought to continue to use the effective request URI for future requests. The server SHOULD generate a Location header field in the response containing a URI reference for the different URI. The user agent MAY use the Location field value for automatic redirection. (Source)

In other words, whenever the server sends a status code of `302`, it says to the client, "Hey, at the moment, the thing you are looking for can be found at this other location."

A key phrase in the specification is "MAY use the Location field value for automatic redirection." It means that you can't force the client to load another URL. The client can choose to wait for user confirmation or decide not to load the URL at all.

Now you know that a redirect is just an HTTP response with a `3xx` status code and a `Location` header. The key takeaway here is that an HTTP redirect is like any old HTTP response, but with an empty body, 3xx status code, and a `Location` header.

That's it. We'll tie this back into Django momentarily, but first let's take a look at two types of redirects in that `3xx` status code range and see why they matter when it comes to web development.

Temporary vs. Permanent Redirects

The HTTP standard specifies several redirect status codes, all in the `3xx` range. The two most common status codes are `301 Permanent Redirect` and `302 Found`.

A status code `302 Found` indicates a temporary redirect. A temporary redirect says, "At the moment, the thing you're looking for can be found at this other address." Think of it like a store sign that reads, "Our store is currently closed for renovation. Please go to our other store around the corner." As this is only temporary, you'd check the original address the next time you go shopping.

> **Note:** In HTTP 1.0, the message for status code 302 was `Temporary Redirect`. The message was changed to `Found` in HTTP 1.1.

As the name implies, permanent redirects are supposed to be permanent. A permanent redirect tells the browser, "The thing you're looking for is no longer at this address. It's now at this new address, and it will never be at the old address again."

A permanent redirect is like a store sign that reads, "We moved. Our new store is just around the corner." This change is permanent, so the next time you want to go to the store, you'd go straight to the new address.

> **Note:** Permanent redirects can have unintended consequences. Finish this guide before using a permanent redirect or jump straight to the section "Permanent redirects are permanent."

Browsers behave similarly when handling redirects: when a URL returns a permanent redirect response, this response is cached. The next time the browser encounters the old URL, it remembers the redirect and directly requests the new address.

Caching a redirect saves an unnecessary request and makes for a better and faster user experience.

Furthermore, the distinction between temporary and permanent redirects is relevant for Search Engine Optimization.

Redirects in Django

Now you know that a redirect is just an HTTP response with a `3xx` status code and a `Location` header.

You could build such a response yourself from a regular `HttpResponse` object:

Python

```
def hand_crafted_redirect_view(request):
  response = HttpResponse(status=302)
  response['Location'] = '/redirect/success/'
  return response
```

This solution is technically correct, but it involves quite a bit of typing.

The `HTTPResponseRedirect` Class

You can save yourself some typing with the class `HttpResponseRedirect`, a subclass of `HttpResponse`. Just instantiate the class with the URL you want to redirect to as the first argument, and the class will set the correct status and Location header:

Python

```python
def redirect_view(request):
  return HttpResponseRedirect('/redirect/success/')
```

You can play with the `HttpResponseRedirect` class in the Python shell to see what you're getting:

Python

```
>>> from django.http import HttpResponseRedirect
>>> redirect = HttpResponseRedirect('/redirect/success/')
>>> redirect.status_code
302
>>> redirect['Location']
'/redirect/success/'
```

There is also a class for permanent redirects, which is aptly named `HttpResponsePermanentRedirect`. It works the same as `HttpResponseRedirect`, the only difference is that it has a status code of `301 (Moved Permanently)`.

> **Note:** In the examples above, the redirect URLs are hard-coded. Hard-coding URLs is bad practice: if the URL ever changes, you have to search through all your code and change any occurrences. Let's fix that!

You could use `django.urls.reverse()` to build a URL, but there is a more convenient way as you will see in the next section.

The `redirect()` Function

To make your life easier, Django provides the versatile shortcut function you've already seen in the introduction: `django.shortcuts.redirect()`.

You can call this function with:

- A model instance, or any other object, with a `get_absolute_url()` method
- A URL or view name and positional and/or keyword arguments
- A URL

It will take the appropriate steps to turn the arguments into a URL and return an `HTTPResponseRedirect`. If you pass `permanent=True`, it will return an instance of `HttpResponsePermanentRedirect`, resulting in a permanent redirect.

Here are three examples to illustrate the different use cases:

1. Passing a model:

   Python

   ```python
   from django.shortcuts import redirect

   def model_redirect_view(request):
       product = Product.objects.filter(featured=True).first()
       return redirect(product)
   ```

   `redirect()` will call `product.get_absolute_url()` and use the result as redirect target. If the given class, in this case `Product`, doesn't have a `get_absolute_url()` method, this will fail with a `TypeError`.

2. Passing a URL name and arguments:

   Python

   ```python
   from django.shortcuts import redirect

   def fixed_featured_product_view(request):
       ...
       product_id = settings.FEATURED_PRODUCT_ID
       return redirect('product_detail', product_id=product_id)
   ```

`redirect()` will try to use its given arguments to reverse a URL. This example assumes your URL patterns contain a pattern like this:

```
path('/product/<product_id>/', 'product_detail_view', name='product_detail')
```

3. Passing a URL:

Python

```
from django.shortcuts import redirect

def featured_product_view(request):
    return redirect('/products/42/')
```

`redirect()` will treat any string containing a `/` or `.` as a URL and use it as redirect target.

The `RedirectView` Class-Based View

If you have a view that does nothing but returning a redirect, you could use the class-based view `django.views.generic.base.RedirectView`.

You can tailor `RedirectView` to your needs through various attributes.

If the class has a `.url` attribute, it will be used as a redirect URL. String formatting placeholders are replaced with named arguments from the URL:

Python

```
# urls.py
from django.urls import path
from .views import SearchRedirectView

urlpatterns = [
    path('/search/<term>/', SearchRedirectView.as_view())
]

# views.py
from django.views.generic.base import RedirectView

class SearchRedirectView(RedirectView):
  url = 'https://google.com/?q=%(term)s'
```

The URL pattern defines an argument `term`, which is used in `SearchRedirectView` to build the redirect URL. The path `/search/kittens/` in your application will redirect you to `https://google.com/?q=kittens`.

Instead of subclassing `RedirectView` to overwrite the `url` attribute, you can also pass the keyword argument `url` to `as_view()` in your `urlpatterns`:

Python

```
#urls.py
from django.views.generic.base import RedirectView

urlpatterns = [
    path('/search/<term>/',
         RedirectView.as_view(url='https://google.com/?q=%(term)s')),
]
```

You can also overwrite `get_redirect_url()` to get a completely custom behavior:

Python

```
from random import choice
from django.views.generic.base import RedirectView

class RandomAnimalView(RedirectView):

     animal_urls = ['/dog/', '/cat/', '/parrot/']
     is_permanent = True

     def get_redirect_url(*args, **kwargs):
        return choice(self.animal_urls)
```

This class-based view redirects to a URL picked randomly from `.animal_urls`.

`django.views.generic.base.RedirectView` offers a few more hooks for customization. Here is the complete list:

- `.url`

  If this attribute is set, it should be a string with a URL to redirect to. If it contains string formatting placeholders like `%(name)s`, they are expanded using the keyword arguments passed to the view.

- `.pattern_name`

  If this attribute is set, it should be the name of a URL pattern to redirect to. Any positional and keyword arguments passed to the view are used to reverse the URL pattern.

- `.permanent`

  If this attribute is `True`, the view returns a permanent redirect. It defaults to `False`.

- `.query_string`

  If this attribute is `True`, the view appends any provided query string to the redirect URL. If it is `False`, which is the default, the query string is discarded.

- `get_redirect_url(*args, **kwargs)`

  This method is responsible for building the redirect URL. If this method returns `None`, the view returns a `410 Gone` status.

  The default implementation first checks `.url`. It treats `.url` as an "old-style" format string, using any named URL parameters passed to the view to expand any named format specifiers.

  If `.url` is not set, it checks if `.pattern_name` is set. If it is, it uses it to reverse a URL with any positional and keyword arguments it received.

  You can change that behavior in any way you want by overwriting this method. Just make sure it returns a string containing a URL.

> **Note:** Class-based views are a powerful concept but can be a bit difficult to wrap your head around. Unlike regular function-based views, where it's relatively straightforward to follow the flow of the code, class-based views are made up of a complex hierarchy of mixins and base classes.
>
> A great tool to make sense of a class-based view class is the website Classy Class-Based Views.

You could implement the functionality of `RandomAnimalView` from the example above with this simple function-based view:

Python

```
from random import choice
from django.shortcuts import redirect

def random_animal_view(request):
    animal_urls = ['/dog/', '/cat/', '/parrot/']
    return redirect(choice(animal_urls))
```

As you can see, the class-based approach does not provide any obvious benefit while adding some hidden complexity. That raises the question: when should you use `RedirectView`?

If you want to add a redirect directly in your `urls.py`, using `RedirectView` makes sense. But if you find yourself overwriting `get_redirect_url`, a function-based view might be easier to understand and more flexible for future enhancements.

Advanced Usage

Once you know that you probably want to use `django.shortcuts.redirect()`, redirecting to a different URL is quite straightforward. But there are a couple of advanced use cases that are not so obvious.

Passing Parameters with Redirects

Sometimes, you want to pass some parameters to the view you're redirecting to. Your best option is to pass the data in the query string of your redirect URL, which means redirecting to a URL like this:

```
http://example.com/redirect-path/?parameter=value
```

Let's assume you want to redirect from `some_view()` to `product_view()`, but pass an optional parameter `category`:

Python

```python
from django.urls import reverse
from urllib.parse import urlencode

def some_view(request):
    ...
    base_url = reverse('product_view')  # 1 /products/
    query_string =  urlencode({'category': category.id})  # 2 category=42
    url = '{}?{}'.format(base_url, query_string)  # 3 /products/?category=42
    return redirect(url)  # 4

def product_view(request):
    category_id = request.GET.get('category')  # 5
    # Do something with category_id
```

The code in this example is quite dense, so let's follow it step by step:

1. First, you use `django.urls.reverse()` to get the URL mapping to `product_view()`.
2. Next, you have to build the query string. That's the part after the question mark. It's advisable to use `urllib.urlparse.urlencode()` for that, as it will take care of properly encoding any special characters.
3. Now you have to join `base_url` and `query_string` with a question mark. A format string works fine for that.
4. Finally, you pass `url` to `django.shortcuts.redirect()` or to a redirect response class.
5. In `product_view()`, your redirect target, the parameter will be available in the `request.GET` dictionary. The parameter might be missing, so you should use `requests.GET.get('category')` instead of `requests.GET['category']`. The former returns `None` when the parameter does not exist, while the latter would raise an exception.

Note: Make sure to validate any data you read from query strings. It might seem like this data is under your control because you created the redirect URL.

In reality, the redirect could be manipulated by the user and must not be trusted, like any other user input. Without proper validation, an attacker might be able gain unauthorized access.

Special Redirect Codes

Django provides HTTP response classes for the status codes `301` and `302`. Those should cover most use cases, but if you ever have to return status codes `303`, `307`, or `308`, you can quite easily create your own response class. Simply subclass `HttpResponseRedirectBase` and overwrite the `status_code` attribute:

Python

```
class HttpResponseTemporaryRedirect(HttpResponseRedirectBase):
    status_code = 307
```

Alternatively, you can use the `django.shortcuts.redirect()` method to create a response object and change the return value. This approach makes sense when you have the name of a view or URL or a model you want to redirect to:

Python

```
def temporary_redirect_view(request):
    response = redirect('success_view')
    response.status_code = 307
    return response
```

Note: There is actually a third class with a status code in the `3xx` range: `HttpResponseNotModified`, with the status code `304`. It indicates that the content URL has not changed and that the client can use a cached version.

One could argue that `304 Not Modified` response redirects to the cached version of a URL, but that's a bit of a stretch. Consequently, it is no longer listed in the "Redirection 3xx" section of the HTTP standard.

Pitfalls

Redirects That Just Won't Redirect

The simplicity of `django.shortcuts.redirect()` can be deceiving. The function itself doesn't perform a redirect: it just returns a redirect response object. You must return this response object from your view (or in a middleware). Otherwise, no redirect will happen.

But even if you know that just calling `redirect()` is not enough, it's easy to introduce this bug into a working application through a simple refactoring. Here's an example to illustrate that.

Let's assume you are building a shop and have a view that is responsible for displaying a product. If the product does not exist, you redirect to the homepage:

Python

```
def product_view(request, product_id):
    try:
        product = Product.objects.get(pk=product_id)
    except Product.DoesNotExist:
        return redirect('/')
    return render(request, 'product_detail.html', {'product': product})
```

Now you want to add a second view to display customer reviews for a product. It should also redirect to the homepage for non-existing products, so as a first step, you extract this functionality from `product_view()` into a helper function `get_product_or_redirect()`:

Python

```python
def get_product_or_redirect(product_id):
    try:
        return Product.objects.get(pk=product_id)
    except Product.DoesNotExist:
        return redirect('/')

def product_view(request, product_id):
    product = get_product_or_redirect(product_id)
    return render(request, 'product_detail.html', {'product': product})
```

Unfortunately, after the refactoring, the redirect does not work anymore.

Can you spot the error? Show/Hide

Redirects That Just Won't Stop Redirecting

When dealing with redirects, you might accidentally create a redirect loop, by having URL A return a redirect that points to URL B which returns a redirect to URL A, and so on. Most HTTP clients detect this kind of redirect loop and will display an error message after a number of requests.

Unfortunately, this kind of bug can be tricky to spot because everything looks fine on the server side. Unless your users complain about the issue, the only indication that something might be wrong is that you've got a number of requests from one client that all result in a redirect response in quick succession, but no response with a `200 OK` status.

Here's a simple example of a redirect loop:

Python

```python
def a_view(request):
    return redirect('another_view')

def another_view(request):
    return redirect('a_view')
```

This example illustrates the principle, but it's overly simplistic. The redirect loops you'll encounter in real-life are probably going to be harder to spot. Let's look at a more elaborate example:

Python

```python
def featured_products_view(request):
    featured_products = Product.objects.filter(featured=True)
    if len(featured_products == 1):
        return redirect('product_view', kwargs={'product_id': featured_products[0].id})
    return render(request, 'featured_products.html', {'product': featured_products})

def product_view(request, product_id):
    try:
        product = Product.objects.get(pk=product_id, in_stock=True)
    except Product.DoesNotExist:
        return redirect('featured_products_view')
    return render(request, 'product_detail.html', {'product': product})
```

`featured_products_view()` fetches all featured products, in other words `Product` instances with `.featured` set to `True`. If only one featured product exists, it redirects directly to `product_view()`. Otherwise, it renders a template with the `featured_products` queryset.

The `product_view` looks familiar from the previous section, but it has two minor differences:

- The view tries to fetch a `Product` that is in stock, indicated by having `.in_stock` set to `True`.
- The view redirects to `featured_products_view()` if no product is in stock.

This logic works fine until your shop becomes a victim of its own success and the one featured product you currently have goes out of stock. If you set `.in_stock` to `False` but forget to set `.featured` to `False` as well, then any visitor to your `feature_product_view()` will now be stuck in a redirect loop.

There is no bullet-proof way to prevent this kind of bug, but a good starting point is to check if the view you are redirecting to uses redirects itself.

Permanent Redirects Are Permanent

Permanent redirects can be like bad tattoos: they might seem like a good idea at the time, but once you realize they were a mistake, it can be quite hard to get rid of them.

When a browser receives a permanent redirect response for a URL, it caches this response indefinitely. Any time you request the old URL in the future, the browser doesn't bother loading it and directly loads the new URL.

It can be quite tricky to convince a browser to load a URL that once returned a permanent redirect. Google Chrome is especially aggressive when it comes to caching redirects.

Why can this be a problem?

Imagine you want to build a web application with Django. You register your domain at `myawesomedjangowebapp.com`. As a first step, you install a blog app at `https://myawesomedjangowebapp.com/blog/` to build a launch mailing list.

Your site's homepage at `https://myawesomedjangowebapp.com/` is still under construction, so you redirect to `https://myawesomedjangowebapp.com/blog/`. You decide to use a permanent redirect because you heard that permanent redirects are cached and caching make things faster, and faster is better because speed is a factor for ranking in Google search results.

As it turns out, you're not only a great developer, but also a talented writer. Your blog becomes popular, and your launch mailing list grows. After a couple of months, your app is ready. It now has a shiny homepage, and you finally remove the redirect.

You send out an announcement email with a special discount code to your sizeable launch mailing list. You lean back and wait for the sign-up notifications to roll in.

To your horror, your mailbox fills with messages from confused visitors who want to visit your app but are always being redirected to your blog.

What has happened? Your blog readers had visited `https://myawesomedjangowebapp.com/` when the redirect to `https://myawesomedjangowebapp.com/blog/` was still active. Because it was a permanent redirect, it was cached in their browsers.

When they clicked on the link in your launch announcement mail, their browsers never bothered to check your new homepage and went straight to your blog. Instead of celebrating your successful launch, you're busy instructing your users how to fiddle with `chrome://net-internals` to reset the cache of their browsers.

The permanent nature of permanent redirects can also bite you while developing on your local machine. Let's rewind to the moment when you implemented that fateful permanent redirect for myawesomedjangowebapp.com.

You start the development server and open `http://127.0.0.1:8000/`. As intended, your app redirects your browser to `http://127.0.0.1:8000/blog/`. Satisfied with your work, you stop the development server and go to lunch.

You return with a full belly, ready to tackle some client work. The client wants some simple changes to their homepage, so you load the client's project and start the development server.

But wait, what is going on here? The homepage is broken, it now returns a 404! Due to the afternoon slump, it takes you a while to notice that you're being redirected to `http://127.0.0.1:8000/blog/`, which doesn't exist in the client's project.

To the browser, it doesn't matter that the URL `http://127.0.0.1:8000/` now serves a completely different application. All that matters to the browser is that this URL once in the past returned a permanent redirect to `http://127.0.0.1:8000/blog/`.

The takeaway from this story is that you should only use permanent redirects on URLs that you've no intention of ever using again. There is a place for permanent redirects, but you must be aware of their consequences.

Even if you're confident that you really need a permanent redirect, it's a good idea to implement a temporary redirect first and only switch to its permanent cousin once you're 100% sure everything works as intended.

Unvalidated Redirects Can Compromise Security

From a security perspective, redirects are a relatively safe technique. An attacker cannot hack a website with a redirect. After all, a redirect just redirects to a URL that an attacker could just type in the address bar of their browser.

However, if you use some kind of user input, like a URL parameter, without proper validation as a redirect URL, this could be abused by an attacker for a phishing attack. This kind of redirect is called an open or unvalidated redirect.

There are legitimate use cases for redirecting to URL that is read from user input. A prime example is Django's login view. It accepts a URL parameter `next` that contains the URL of the page the user is redirected to after login. To redirect the user to their profile after login, the URL might look like this:

```
https://myawesomedjangowebapp.com/login/?next=/profile/
```

Django does validate the `next` parameter, but let's assume for a second that it doesn't.

Without validation, an attacker could craft a URL that redirects the user to a website under their control, for example:

```
https://myawesomedjangowebapp.com/login/?next=https://myawesomedjangowebapp.co/profile/
```

The website `myawesomedjangowebapp.co` might then display an error message and trick the user into entering their credentials again.

The best way to avoid open redirects is to not use any user input when building a redirect URL.

If you cannot be sure that a URL is safe for redirection, you can use the function `django.utils.http.is_safe_url()` to validate it. The docstring explains its usage quite well:

> `is_safe_url(url, host=None, allowed_hosts=None, require_https=False)`
>
> Return `True` if the url is a safe redirection (i.e. it doesn't point to a different host and uses a safe scheme). Always return `False` on an empty url. If `require_https` is `True`, only 'https' will be considered a valid scheme, as opposed to 'http' and 'https' with the default, `False`. (Source)

Let's look at some examples.

A relative URL is considered safe:

Python

```
>>> # Import the function first.
>>> from django.utils.http import is_safe_url
>>> is_safe_url('/profile/')
True
```

A URL pointing to another host is generally not considered safe:

Python

```
>>> is_safe_url('https://myawesomedjangowebapp.com/profile/')
False
```

A URL pointing to another host is considered safe if its host is provided in `allowed_hosts`:

Python

```
>>> is_safe_url('https://myawesomedjangowebapp.com/profile/',
...             allowed_hosts={'myawesomedjangowebapp.com'})
True
```

If the argument `require_https` is `True`, a URL using the `http` scheme is not considered safe:

Python

```
>>> is_safe_url('http://myawesomedjangowebapp.com/profile/',
...             allowed_hosts={'myawesomedjangowebapp.com'},
...             require_https=True)
False
```


Summary

This wraps up this guide on HTTP redirects with Django. Congratulations: you have now touched on every aspect of redirects all the way from the low-level details of the HTTP protocol to the high-level way of dealing with them in Django.

You learned how an HTTP redirect looks under the hood, what the different status codes are, and how permanent and temporary redirects differ. This knowledge is not specific to Django and is valuable for web development in any language.

You can now perform a redirect with Django, either by using the redirect response classes `HttpResponseRedirect` and `HttpResponsePermanentRedirect`, or with the convenience function `django.shortcuts.redirect()`. You saw solutions for a couple of advanced use cases and know how to steer clear of common pitfalls.

If you have any further question about HTTP redirects leave a comment below and in the meantime, happy redirecting!

References

- Django documentation: `django.http.HttpResponseRedirect`
- Django documentation: `django.shortcuts.render()`
- Django documentation: `django.views.generic.base.RedirectView`
- RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content - 6.4 Redirection 3xx
- CWE-601: URL Redirection to Untrusted Site ('Open Redirect')