Django

Documentation

Request and response objects

Quick overview

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an **`HttpRequest`** object that contains metadata about the request. Then Django loads the appropriate view, passing the **`HttpRequest`** as the first argument to the view function. Each view is responsible for returning an **`HttpResponse`** object.

This document explains the APIs for **`HttpRequest`** and **`HttpResponse`** objects, which are defined in the **`django.http`** module.

`HttpRequest` objects

class **`HttpRequest`[source]**

Attributes

All attributes should be considered read-only, unless stated otherwise.

`HttpRequest.scheme`

A string representing the scheme of the request (**`http`** or **`https`** usually).

`HttpRequest.body`

The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use **`HttpRequest.POST`**.

You can also read from an HttpRequest using a file-like interface. See **`HttpRequest.read()`**.

`HttpRequest.path`

A string representing the full path to the requested page, not including the scheme or domain.

Example: **`"/music/bands/the_beatles/"`**

`HttpRequest.path_info`

Under some Web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The **`path_info`** attribute always contains the path info portion of the path, no matter what Web server is being used. Using this instead of **`path`** can make your code easier to move between test and deployment servers.

For example, if the **`WSGIScriptAlias`** for your application is set to **`"/minfo"`**, then **`path`** might be **`"/minfo/music/bands/the_beatles/"`** and **`path_info`** would be **`"/music/bands/the_beatles/"`**.

`HttpRequest.method`

A string representing the HTTP method used in the request. This is guaranteed to be uppercase. Example:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

Language: **en**

Documentation version: **1.10**

`HttpRequest.encoding`

A string representing the current encoding used to decode form submission data (or **None**, which means the **DEFAULT_CHARSET** setting is used). You can write to this attribute to change the encoding used when accessing the form data. Any subsequent attribute accesses (such as reading from **GET** or **POST**) will use the new **encoding** value. Useful if you know the form data is not in the **DEFAULT_CHARSET** encoding.

HttpRequest.content_type

New in Django 1.10.

A string representing the MIME type of the request, parsed from the **CONTENT_TYPE** header.

HttpRequest.content_params

New in Django 1.10.

A dictionary of key/value parameters included in the **CONTENT_TYPE** header.

HttpRequest.GET

A dictionary-like object containing all given HTTP GET parameters. See the **QueryDict** documentation below.

HttpRequest.POST

A dictionary-like object containing all given HTTP POST parameters, providing that the request contains form data. See the **QueryDict** documentation below. If you need to access raw or non-form data posted in the request, access this through the **HttpRequest.body** attribute instead.

It's possible that a request can come in via POST with an empty **POST** dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn't use **if request.POST** to check for use of the POST method; instead, use **if request.method == "POST"** (see above).

Note: **POST** does *not* include file-upload information. See **FILES**.

HttpRequest.COOKIES

A standard Python dictionary containing all cookies. Keys and values are strings.

HttpRequest.FILES

A dictionary-like object containing all uploaded files. Each key in **FILES** is the **name** from the **<input type="file" name="" />**. Each value in **FILES** is an **UploadedFile**.

See Managing files for more information.

Note that **FILES** will only contain data if the request method was POST and the **<form>** that posted to the request had **enctype="multipart/form-data"**. Otherwise, **FILES** will be a blank dictionary-like object.

HttpRequest.META

A standard Python dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

- **CONTENT_LENGTH** – The length of the request body (as a string).
- **CONTENT_TYPE** – The MIME type of the request body.
- **HTTP_ACCEPT** – Acceptable content types for the response.
- **HTTP_ACCEPT_ENCODING** – Acceptable encodings for the response.
- **HTTP_ACCEPT_LANGUAGE** – Acceptable languages for the response.
- **HTTP_HOST** – The HTTP Host header sent by the client.
- **HTTP_REFERER** – The referring page, if any.
- **HTTP_USER_AGENT** – The client's user-agent string.
- **QUERY_STRING** – The query string, as a single (unparsed) string.
- **REMOTE_ADDR** – The IP address of the client.
- **REMOTE_HOST** – The hostname of the client.
- **REMOTE_USER** – The user authenticated by the Web server, if any.
- **REQUEST_METHOD** – A string such as **"GET"** or **"POST"**.
- **SERVER_NAME** – The hostname of the server.

- **SERVER_PORT** – The port of the server (as a string).

With the exception of **CONTENT_LENGTH** and **CONTENT_TYPE**, as given above, any HTTP headers in the request are converted to **META** keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an **HTTP_** prefix to the name. So, for example, a header called **X-Bender** would be mapped to the **META** key **HTTP_X_BENDER**.

Note that **runserver** strips all headers with underscores in the name, so you won't see them in **META**. This prevents header-spoofing based on ambiguity between underscores and dashes both being normalizing to underscores in WSGI environment variables. It matches the behavior of Web servers like Nginx and Apache 2.4+.

HttpRequest.resolver_match

An instance of **ResolverMatch** representing the resolved URL. This attribute is only set after URL resolving took place, which means it's available in all views but not in middleware which are executed before URL resolving takes place (you can use it in **process_view()** though).

Attributes set by application code

Django doesn't set these attributes itself but makes use of them if set by your application.

HttpRequest.current_app

The **url** template tag will use its value as the **current_app** argument to **reverse()**.

HttpRequest.urlconf

This will be used as the root URLconf for the current request, overriding the **ROOT_URLCONF** setting. See How Django processes a request for details.

urlconf can be set to **None** to revert any changes made by previous middleware and return to using the **ROOT_URLCONF**.

> **Changed in Django 1.9:**
> Setting **urlconf=None** raised **ImproperlyConfigured** in older versions.

Attributes set by middleware

Some of the middleware included in Django's contrib apps set attributes on the request. If you don't see the attribute on a request, be sure the appropriate middleware class is listed in **MIDDLEWARE**.

HttpRequest.session

From the **SessionMiddleware**: A readable and writable, dictionary-like object that represents the current session.

HttpRequest.site

From the **CurrentSiteMiddleware**: An instance of **Site** or **RequestSite** as returned by **get_current_site()** representing the current site.

HttpRequest.user

From the **AuthenticationMiddleware**: An instance of **AUTH_USER_MODEL** representing the currently logged-in user. If the user isn't currently logged in, **user** will be set to an instance of **AnonymousUser**. You can tell them apart with **is_authenticated**, like so:

```
if request.user.is_authenticated:
    ... # Do something for logged-in users.
else:
    ... # Do something for anonymous users.
```

Methods

HttpRequest.get_host()[source]

Returns the originating host of the request using information from the **HTTP_X_FORWARDED_HOST** (if **USE_X_FORWARDED_HOST** is enabled) and **HTTP_HOST** headers, in that order. If they don't provide a value, the method uses a combination of **SERVER_NAME** and **SERVER_PORT** as detailed in **PEP 3333**.

Example: **"127.0.0.1:8000"**

Note

The **get_host()** method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

```
from django.utils.deprecation import MiddlewareMixin

class MultipleProxyMiddleware(MiddlewareMixin):
    FORWARDED_FOR_FIELDS = [
        'HTTP_X_FORWARDED_FOR',
        'HTTP_X_FORWARDED_HOST',
        'HTTP_X_FORWARDED_SERVER',
    ]

    def process_request(self, request):
        """
        Rewrites the proxy headers so that only the most
        recent proxy is used.
        """
        for field in self.FORWARDED_FOR_FIELDS:
            if field in request.META:
                if ',' in request.META[field]:
                    parts = request.META[field].split(',')
                    request.META[field] = parts[-1].strip()
```

This middleware should be positioned before any other middleware that relies on the value of **get_host()** – for instance, **CommonMiddleware** or **CsrfViewMiddleware**.

HttpRequest.get_port()[source]

New in Django 1.9.

Returns the originating port of the request using information from the **HTTP_X_FORWARDED_PORT** (if **USE_X_FORWARDED_PORT** is enabled) and **SERVER_PORT META** variables, in that order.

HttpRequest.get_full_path()[source]

Returns the **path**, plus an appended query string, if applicable.

Example: **"/music/bands/the_beatles/?print=true"**

HttpRequest.build_absolute_uri(*location*)[source]

Returns the absolute URI form of **location**. If no location is provided, the location will be set to **request.get_full_path()**.

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request.

Example: **"https://example.com/music/bands/the_beatles/?print=true"**

Note

Mixing HTTP and HTTPS on the same site is discouraged, therefore **build_absolute_uri()** will always generate an absolute URI with the same scheme the current request has. If you need to redirect users to HTTPS, it's best to let your Web server redirect all HTTP traffic to HTTPS.

HttpRequest.get_signed_cookie(*key*, *default=RAISE_ERROR*, *salt=''*, *max_age=None*)[source]

Returns a cookie value for a signed cookie, or raises a **django.core.signing.BadSignature** exception if the signature is no longer valid. If you provide the **default** argument the exception will be suppressed and that default value will be returned instead.

The optional **salt** argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the **max_age** argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than **max_age** seconds.

For example:

```
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # assuming cookie was set using the same salt
>>> request.get_signed_cookie('non-existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

See cryptographic signing for more information.

`HttpRequest.is_secure()`[source]

Returns **`True`** if the request is secure; that is, if it was made with HTTPS.

`HttpRequest.is_ajax()`[source]

Returns **`True`** if the request was made via an **`XMLHttpRequest`**, by checking the **`HTTP_X_REQUESTED_WITH`** header for the string **`'XMLHttpRequest'`**. Most modern JavaScript libraries send this header. If you write your own XMLHttpRequest call (on the browser side), you'll have to set this header manually if you want **`is_ajax()`** to work.

If a response varies on whether or not it's requested via AJAX and you are using some form of caching like Django's **`cache middleware`**, you should decorate the view with **`vary_on_headers('HTTP_X_REQUESTED_WITH')`** so that the responses are properly cached.

`HttpRequest.read(`*size=None*`)`[source]

`HttpRequest.readline()`[source]

`HttpRequest.readlines()`[source]

`HttpRequest.xreadlines()`[source]

`HttpRequest.__iter__()`

Methods implementing a file-like interface for reading from an HttpRequest instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with an iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an HttpRequest instance can be passed directly to an XML parser such as ElementTree:

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

`QueryDict` objects

class `QueryDict`[source]

In an **`HttpRequest`** object, the **`GET`** and **`POST`** attributes are instances of **`django.http.QueryDict`**, a dictionary-like class customized to deal with multiple values for the same key. This is necessary because some HTML form elements, notably **`<select multiple>`**, pass multiple values for the same key.

The **`QueryDict`**s at **`request.POST`** and **`request.GET`** will be immutable when accessed in a normal request/response cycle. To get a mutable version you need to use **`.copy()`**.

Methods

QueryDict implements all the standard dictionary methods because it's a subclass of dictionary. Exceptions are outlined here:

QueryDict.__init__(*query_string=None*, *mutable=False*, *encoding=None*)**[source]**

Instantiates a **QueryDict** object based on **query_string**.

```
>>> QueryDict('a=1&a=2&c=3')
<QueryDict: {'a': ['1', '2'], 'c': ['3']}>
```

If **query_string** is not passed in, the resulting **QueryDict** will be empty (it will have no keys or values).

Most **QueryDict**s you encounter, and in particular those at **request.POST** and **request.GET**, will be immutable. If you are instantiating one yourself, you can make it mutable by passing **mutable=True** to its **__init__()**.

Strings for setting both keys and values will be converted from **encoding** to unicode. If encoding is not set, it defaults to **DEFAULT_CHARSET**.

QueryDict.__getitem__(*key*)

Returns the value for the given key. If the key has more than one value, **__getitem__()** returns the last value. Raises **django.utils.datastructures.MultiValueDictKeyError** if the key does not exist. (This is a subclass of Python's standard **KeyError**, so you can stick to catching **KeyError**.)

QueryDict.__setitem__(*key*, *value*)**[source]**

Sets the given key to **[value]** (a Python list whose single element is **value**). Note that this, as other dictionary functions that have side effects, can only be called on a mutable **QueryDict** (such as one that was created via **copy()**).

QueryDict.__contains__(*key*)

Returns **True** if the given key is set. This lets you do, e.g., **if "foo" in request.GET**.

QueryDict.get(*key*, *default=None*)

Uses the same logic as **__getitem__()** above, with a hook for returning a default value if the key doesn't exist.

QueryDict.setdefault(*key*, *default=None*)**[source]**

Just like the standard dictionary **setdefault()** method, except it uses **__setitem__()** internally.

QueryDict.update(*other_dict*)

Takes either a **QueryDict** or standard dictionary. Just like the standard dictionary **update()** method, except it *appends* to the current dictionary items rather than replacing them. For example:

```
>>> q = QueryDict('a=1', mutable=True)
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # returns the last
'2'
```

QueryDict.items()

Just like the standard dictionary **items()** method, except this uses the same last-value logic as **__getitem__()**. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

QueryDict.iteritems()

Just like the standard dictionary **iteritems()** method. Like **QueryDict.items()** this uses the same last-value logic as **QueryDict.__getitem__()**.

QueryDict.iterlists()

Like **QueryDict.iteritems()** except it includes all values, as a list, for each member of the dictionary.

QueryDict.values()

Just like the standard dictionary **values()** method, except this uses the same last-value logic as **__getitem__()**. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

QueryDict.itervalues()

Just like **QueryDict.values()**, except an iterator.

In addition, **QueryDict** has the following methods:

QueryDict.copy()[source]

Returns a copy of the object, using **copy.deepcopy()** from the Python standard library. This copy will be mutable even if the original was not.

QueryDict.getlist(*key*, *default=None*)

Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist and no default value was provided. It's guaranteed to return a list of some sort unless the default value provided is not a list.

QueryDict.setlist(*key*, *list_*)[source]

Sets the given key to **list_** (unlike **__setitem__()**).

QueryDict.appendlist(*key*, *item*)[source]

Appends an item to the internal list associated with key.

QueryDict.setlistdefault(*key*, *default_list=None*)[source]

Just like **setdefault**, except it takes a list of values instead of a single value.

QueryDict.lists()

Like **items()**, except it includes all values, as a list, for each member of the dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

QueryDict.pop(*key*)[source]

Returns a list of values for the given key and removes them from the dictionary. Raises **KeyError** if the key does not exist. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.pop('a')
['1', '2', '3']
```

QueryDict.popitem()[source]

Removes an arbitrary member of the dictionary (since there's no concept of ordering), and returns a two value tuple containing the key and a list of all values for the key. Raises **KeyError** when called on an empty dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.popitem()
('a', ['1', '2', '3'])
```

`QueryDict.dict()`

Returns **`dict`** representation of **`QueryDict`**. For every (key, list) pair in **`QueryDict`**, **`dict`** will have (key, item), where item is one element of the list, using same logic as **`QueryDict.__getitem__()`**:

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{'a': '5'}
```

`QueryDict.urlencode(`*safe=None*`)`[source]

Returns a string of the data in query-string format. Example:

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

Optionally, urlencode can be passed characters which do not require encoding. For example:

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

`HttpResponse` objects

class `HttpResponse`[source]

In contrast to **`HttpRequest`** objects, which are created automatically by Django, **`HttpResponse`** objects are your responsibility. Each view you write is responsible for instantiating, populating and returning an **`HttpResponse`**.

The **`HttpResponse`** class lives in the **`django.http`** module.

Usage

Passing strings

Typical usage is to pass the contents of the page, as a string, to the **`HttpResponse`** constructor:

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```


But if you want to add content incrementally, you can use **`response`** as a file-like object:

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

Passing iterators

Finally, you can pass **HttpResponse** an iterator rather than strings. **HttpResponse** will consume the iterator immediately, store its content as a string, and discard it. Objects with a **close()** method such as files and generators are immediately closed.

If you need the response to be streamed from the iterator to the client, you must use the **StreamingHttpResponse** class instead.

Changed in Django 1.10:

Objects with a **close()** method used to be closed when the WSGI server called **close()** on the response.

Setting header fields

To set or remove a header field in your response, treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

Note that unlike a dictionary, **del** doesn't raise **KeyError** if the header field doesn't exist.

For setting the **Cache-Control** and **Vary** header fields, it is recommended to use the **patch_cache_control()** and **patch_vary_headers()** methods from **django.utils.cache**, since these fields can have multiple, comma-separated values. The "patch" methods ensure that other values, e.g. added by a middleware, are not removed.

HTTP header fields cannot contain newlines. An attempt to set a header field containing a newline character (CR or LF) will raise **BadHeaderError**

Telling the browser to treat the response as a file attachment

To tell the browser to treat the response as a file attachment, use the **content_type** argument and set the **Content-Disposition** header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

There's nothing Django-specific about the **Content-Disposition** header, but it's easy to forget the syntax, so we've included it here.

Attributes

HttpResponse.content

A bytestring representing the content, encoded from a Unicode object if necessary.

HttpResponse.charset

A string denoting the charset in which the response will be encoded. If not given at **HttpResponse** instantiation time, it will be extracted from **content_type** and if that is unsuccessful, the **DEFAULT_CHARSET** setting will be used.

HttpResponse.status_code

The **HTTP status code** for the response.

> **Changed in Django 1.9:**
> Unless **reason_phrase** is explicitly set, modifying the value of **status_code** outside the constructor will also modify the value of **reason_phrase**.

HttpResponse.reason_phrase

The HTTP reason phrase for the response.

> **Changed in Django 1.9:**
> **reason_phrase** no longer defaults to all capital letters. It now uses the **HTTP standard's** default reason phrases.
>
> Unless explicitly set, **reason_phrase** is determined by the current value of **status_code**.

HttpResponse.streaming

This is always **False**.

This attribute exists so middleware can treat streaming responses differently from regular responses.

HttpResponse.closed

True if the response has been closed.

Methods

HttpResponse.__init__(*content=''*, *content_type=None*, *status=200*, *reason=None*, *charset=None*)[source]

Instantiates an **HttpResponse** object with the given page content and content type.

content should be an iterator or a string. If it's an iterator, it should return strings, and those strings will be joined together to form the content of the response. If it is not an iterator or a string, it will be converted to a string when accessed.

content_type is the MIME type optionally completed by a character set encoding and is used to fill the HTTP **Content-Type** header. If not specified, it is formed by the **DEFAULT_CONTENT_TYPE** and **DEFAULT_CHARSET** settings, by default: "*text/html; charset=utf-8*".

status is the **HTTP status code** for the response.

reason is the HTTP response phrase. If not provided, a default phrase will be used.

charset is the charset in which the response will be encoded. If not given it will be extracted from **content_type**, and if that is unsuccessful, the **DEFAULT_CHARSET** setting will be used.

HttpResponse.__setitem__(*header*, *value*)

Sets the given header name to the given value. Both **header** and **value** should be strings.

HttpResponse.__delitem__(*header*)

Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.

HttpResponse.__getitem__(*header*)

Returns the value for the given header name. Case-insensitive.

HttpResponse.has_header(*header*)

Returns **True** or **False** based on a case-insensitive check for a header with the given name.

HttpResponse.setdefault(*header*, *value*)

Sets a header unless it has already been set.

HttpResponse.set_cookie(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*)

Sets a cookie. The parameters are the same as in the **Morsel** cookie object in the Python standard library.

- **max_age** should be a number of seconds, or **None** (default) if the cookie should last only as long as the client's browser session. If **expires** is not specified, it will be calculated.
- **expires** should either be a string in the format **"Wdy, DD-Mon-YY HH:MM:SS GMT"** or a **datetime.datetime** object in UTC. If **expires** is a **datetime** object, the **max_age** will be calculated.

- Use **domain** if you want to set a cross-domain cookie. For example, **domain=".lawrence.com"** will set a cookie that is readable by the domains www.lawrence.com, blogs.lawrence.com and calendars.lawrence.com. Otherwise, a cookie will only be readable by the domain that set it.
- Use **httponly=True** if you want to prevent client-side JavaScript from having access to the cookie.

  HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of a client-side script from accessing the protected cookie data.

Warning

Both **RFC 2109** and **RFC 6265** state that user agents should support cookies of at least 4096 bytes. For many browsers this is also the maximum size. Django will not raise an exception if there's an attempt to store a cookie of more than 4096 bytes, but many browsers will not set the cookie correctly.

HttpResponse.set_signed_cookie(*key*, *value*, *salt=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=True*)

Like **set_cookie()**, but cryptographic signing the cookie before setting it. Use in conjunction with **HttpRequest.get_signed_cookie()**. You can use the optional **salt** argument for added key strength, but you will need to remember to pass it to the corresponding **HttpRequest.get_signed_cookie()** call.

HttpResponse.delete_cookie(*key*, *path='/'*, *domain=None*)

Deletes the cookie with the given key. Fails silently if the key doesn't exist.

Due to the way cookies work, **path** and **domain** should be the same values you used in **set_cookie()** – otherwise the cookie may not be deleted.

HttpResponse.write(*content*)[source]

This method makes an **HttpResponse** instance a file-like object.

HttpResponse.flush()

This method makes an **HttpResponse** instance a file-like object.

HttpResponse.tell()[source]

This method makes an **HttpResponse** instance a file-like object.

HttpResponse.getvalue()[source]

Returns the value of **HttpResponse.content**. This method makes an **HttpResponse** instance a stream-like object.

HttpResponse.readable()

New in Django 1.10:
Always **False**. This method makes an **HttpResponse** instance a stream-like object.

HttpResponse.seekable()

New in Django 1.10:
Always **False**. This method makes an **HttpResponse** instance a stream-like object.

HttpResponse.writable()[source]

Always **True**. This method makes an **HttpResponse** instance a stream-like object.

HttpResponse.writelines(*lines*)[source]

Writes a list of lines to the response. Line separators are not added. This method makes an **HttpResponse** instance a stream-like object.

HttpResponse subclasses

Django includes a number of **HttpResponse** subclasses that handle different types of HTTP responses. Like **HttpResponse**, these subclasses live in **django.http**.

class **HttpResponseRedirect**[source]

The first argument to the constructor is required – the path to redirect to. This can be a fully qualified URL (e.g. **`'https://www.yahoo.com/search/'`**), an absolute path with no domain (e.g. **`'/search/'`**), or even a relative path (e.g. **`'search/'`**). In that last case, the client browser will reconstruct the full URL itself according to the current path. See **`HttpResponse`** for other optional constructor arguments. Note that this returns an HTTP status code 302.

`url`

This read-only attribute represents the URL the response will redirect to (equivalent to the **`Location`** response header).

class **`HttpResponsePermanentRedirect`**[source]

Like **`HttpResponseRedirect`**, but it returns a permanent redirect (HTTP status code 301) instead of a "found" redirect (status code 302).

class **`HttpResponseNotModified`**[source]

The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

class **`HttpResponseBadRequest`**[source]

Acts just like **`HttpResponse`** but uses a 400 status code.

class **`HttpResponseNotFound`**[source]

Acts just like **`HttpResponse`** but uses a 404 status code.

class **`HttpResponseForbidden`**[source]

Acts just like **`HttpResponse`** but uses a 403 status code.

class **`HttpResponseNotAllowed`**[source]

Like **`HttpResponse`**, but uses a 405 status code. The first argument to the constructor is required: a list of permitted methods (e.g. **`['GET', 'POST']`**).

class **`HttpResponseGone`**[source]

Acts just like **`HttpResponse`** but uses a 410 status code.

class **`HttpResponseServerError`**[source]

Acts just like **`HttpResponse`** but uses a 500 status code.

Note

If a custom subclass of **`HttpResponse`** implements a **`render`** method, Django will treat it as emulating a **`SimpleTemplateResponse`**, and the **`render`** method must itself return a valid response object.

`JsonResponse` objects

class **`JsonResponse`**(*data*, *encoder=DjangoJSONEncoder*, *safe=True*, *json_dumps_params=None*, ***kwargs*)[source]

An **`HttpResponse`** subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with a couple differences:

Its default **`Content-Type`** header is set to **`application/json`**.

The first parameter, **`data`**, should be a **`dict`** instance. If the **`safe`** parameter is set to **`False`** (see below) it can be any JSON-serializable object.

The **`encoder`**, which defaults to **`django.core.serializers.json.DjangoJSONEncoder`**, will be used to serialize the data. See JSON serialization for more details about this serializer.

The **`safe`** boolean parameter defaults to **`True`**. If it's set to **`False`**, any object can be passed for serialization (otherwise only **`dict`** instances are allowed). If **`safe`** is **`True`** and a non-**`dict`** object is passed as the first argument, a **`TypeError`** will be raised.

The **`json_dumps_params`** parameter is a dictionary of keyword arguments to pass to the **`json.dumps()`** call used to generate the response.

Changed in Django 1.9:

The **`json_dumps_params`** argument was added.

Usage

Typical usage could look like:

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content
b'{"foo": "bar"}'
```

Serializing non-dictionary objects

In order to serialize objects other than **`dict`** you must set the **`safe`** parameter to **`False`**:

```
>>> response = JsonResponse([1, 2, 3], safe=False)
```

Without passing **`safe=False`**, a **`TypeError`** will be raised.

Warning

Before the 5th edition of ECMAScript it was possible to poison the JavaScript **`Array`** constructor. For this reason, Django does not allow passing non-dict objects to the **`JsonResponse`** constructor by default. However, most modern browsers implement EcmaScript 5 which removes this attack vector. Therefore it is possible to disable this security precaution.

Changing the default JSON encoder

If you need to use a different JSON encoder class you can pass the **`encoder`** parameter to the constructor method:

```
>>> response = JsonResponse(data, encoder=MyJSONEncoder)
```

`StreamingHttpResponse` objects

class **`StreamingHttpResponse`**[source]

The **`StreamingHttpResponse`** class is used to stream a response from Django to the browser. You might want to do this if generating the response takes too long or uses too much memory. For instance, it's useful for generating large CSV files.

Performance considerations

Django is designed for short-lived requests. Streaming responses will tie a worker process for the entire duration of the response. This may result in poor performance.

Generally speaking, you should perform expensive tasks outside of the request-response cycle, rather than resorting to a streamed response.

The **StreamingHttpResponse** is not a subclass of **HttpResponse**, because it features a slightly different API. However, it is almost identical, with the following notable differences:

- It should be given an iterator that yields strings as content.
- You cannot access its content, except by iterating the response object itself. This should only occur when the response is returned to the client.
- It has no **content** attribute. Instead, it has a **streaming_content** attribute.
- You cannot use the file-like object **tell()** or **write()** methods. Doing so will raise an exception.

StreamingHttpResponse should only be used in situations where it is absolutely required that the whole content isn't iterated before transferring the data to the client. Because the content can't be accessed, many middlewares can't function normally. For example the **ETag** and **Content-Length** headers can't be generated for streaming responses.

Attributes

StreamingHttpResponse.streaming_content

An iterator of strings representing the content.

StreamingHttpResponse.status_code

The **HTTP status code** for the response.

> **Changed in Django 1.9:**
> Unless **reason_phrase** is explicitly set, modifying the value of **status_code** outside the constructor will also modify the value of **reason_phrase**.

StreamingHttpResponse.reason_phrase

The HTTP reason phrase for the response.

> **Changed in Django 1.9:**
> **reason_phrase** no longer defaults to all capital letters. It now uses the **HTTP standard's** default reason phrases.
>
> Unless explicitly set, **reason_phrase** is determined by the current value of **status_code**.

StreamingHttpResponse.streaming

This is always **True**.

FileResponse objects

class **FileResponse**[source]

FileResponse is a subclass of **StreamingHttpResponse** optimized for binary files. It uses wsgi.file_wrapper if provided by the wsgi server, otherwise it streams the file out in small chunks.

FileResponse expects a file open in binary mode like so:

```
>>> from django.http import FileResponse
>>> response = FileResponse(open('myfile.png', 'rb'))
```

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List