# Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the class-based views reference documentation.

- Introduction to class-based views
- Built-in class-based generic views
- Form handling with class-based views
- Using mixins with class-based views

## Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the `View` class, which handles linking the view into the URLs, HTTP method dispatching and other common features. `RedirectView` provides a HTTP redirect, and `TemplateView` extends the base class to make it also render a template.

## Usage in your URLconf

The most direct way to use generic views is to create them directly in your URLconf. If you're only changing a few attributes on a class-based view, you can pass them into the `as_view()` method call itself:

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

urlpatterns = [
    path("about/", TemplateView.as_view(template_name="about.html")),
]
```

Any arguments passed to `as_view()` will override attributes set on the class. In this example, we set `template_name` on the `TemplateView`. A similar overriding pattern can be used for the `url` attribute on `RedirectView`.

## Subclassing generic views

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the `template_name`) or methods (such as `get_context_data`) in your subclass to provide new values or methods. Consider, for

example, a view that just displays one template, `about.html`. Django has a generic view to do this - `TemplateView` - so we can subclass it, and override the template name:

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


class AboutView(TemplateView):
    template_name = "about.html"
```

Then we need to add this new view into our URLconf. `TemplateView` is a class, not a function, so we point the URL to the `as_view()` class method instead, which provides a function-like entry to class-based views:

```
# urls.py
from django.urls import path
from some_app.views import AboutView

urlpatterns = [
    path("about/", AboutView.as_view()),
]
```

For more information on how to use the built in generic views, consult the next topic on generic class-based views.

### Supporting other HTTP methods

Suppose somebody wants to access our book library over HTTP using the views as an API. The API client would connect every now and then and download book data for the books published since last visit. But if no new books appeared since then, it is a waste of CPU time and bandwidth to fetch the books from the database, render a full response and send it to the client. It might be preferable to ask the API when the most recent book was published.

We map the URL to book list view in the URLconf:

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

urlpatterns = [
    path("books/", BookListView.as_view()),
]
```

And the view:

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book


class BookListView(ListView):
    model = Book
```

```
    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest("publication_date")
        response = HttpResponse(
            # RFC 1123 date format.
            headers={
                "Last-Modified": last_book.publication_date.strftime(
                    "%a, %d %b %Y %H:%M:%S GMT"
                )
            },
        )
        return response
```

If the view is accessed from a `GET` request, an object list is returned in the response (using the `book_list.html` template). But if the client issues a `HEAD` request, the response has an empty body and the `Last-Modified` header indicates when the most recent book was published. Based on this information, the client may or may not download the full object list.

## Asynchronous class-based views

As well as the synchronous ( `def` ) method handlers already shown, `View` subclasses may define asynchronous ( `async def` ) method handlers to leverage asynchronous code using `await` :

```
import asyncio
from django.http import HttpResponse
from django.views import View


class AsyncView(View):
    async def get(self, request, *args, **kwargs):
        # Perform io-blocking view logic using await, sleep for example.
        await asyncio.sleep(1)
        return HttpResponse("Hello async world!")
```

Within a single view-class, all user-defined method handlers must be either synchronous, using `def` , or all asynchronous, using `async def` . An `ImproperlyConfigured` exception will be raised in `as_view()` if `def` and `async def` declarations are mixed.

Django will automatically detect asynchronous views and run them in an asynchronous context. You can read more about Django's asynchronous support, and how to best use async views, in Asynchronous support.