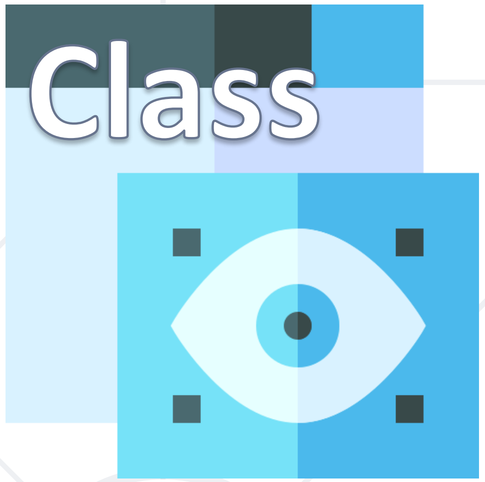
Class-Based-Views

Create Views Without Having to Write Too Much Code



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Table of Content

Have a Question?

Software University

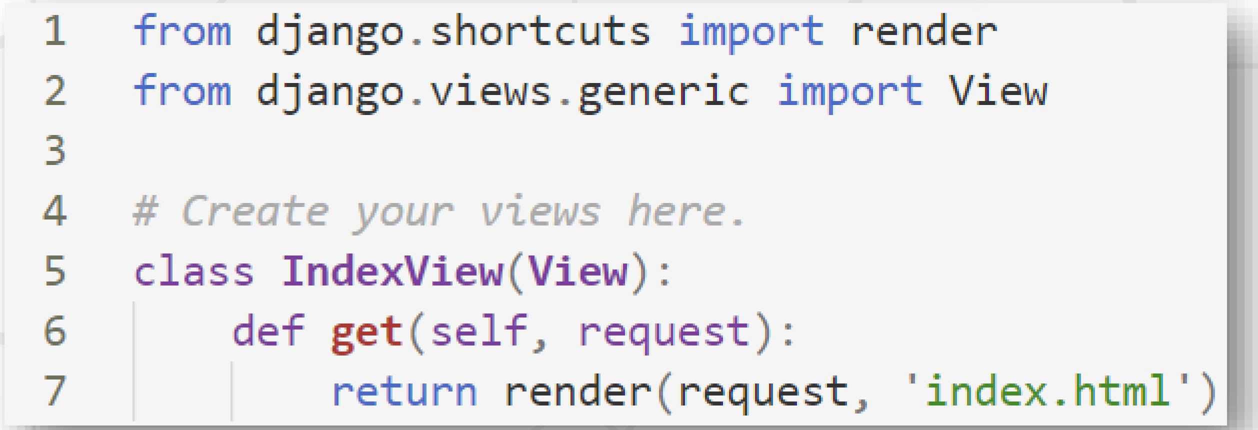
sli.do

#python-web

What are Class-Based Views?

What are CBV's?

- A view is a callable which takes a **request** and returns a **response**
- Class-based views provide an alternative way to implement views as Python **objects** instead of **functions**



```python
from django.shortcuts import render
from django.views.generic import View

# Create your views here.
class IndexView(View):
    def get(self, request):
        return render(request, 'index.html')
```

CBV's Inheritance Structure

- Class-Based-Views use class **inheritance**
- They also use the "**mixin**" pattern
  - You can create classes with related functionality
  - You can include that class as parent of another class

CBV vs. FBV - Pros

- **Class-Based Views**
  - Easily extended
  - Can use techniques like mixins
  - Handling HTTP methods in separate class methods
  - Built-in generic CBV's
- **Function-Based Views**
  - Simple to implement
  - Easy to read
  - Explicit code flow
  - Straightforward usage of decorators

CBV vs. FBV - Cons

- **Class-Based Views**
  - Harder to read
  - Implicit code flow
  - Hidden code in parent classes, mixins
  - Use of decorators require extra import
- **Function-Based Views**
  - Hard to extend
  - Hard to reuse
  - Handling HTTP methods via conditional branching

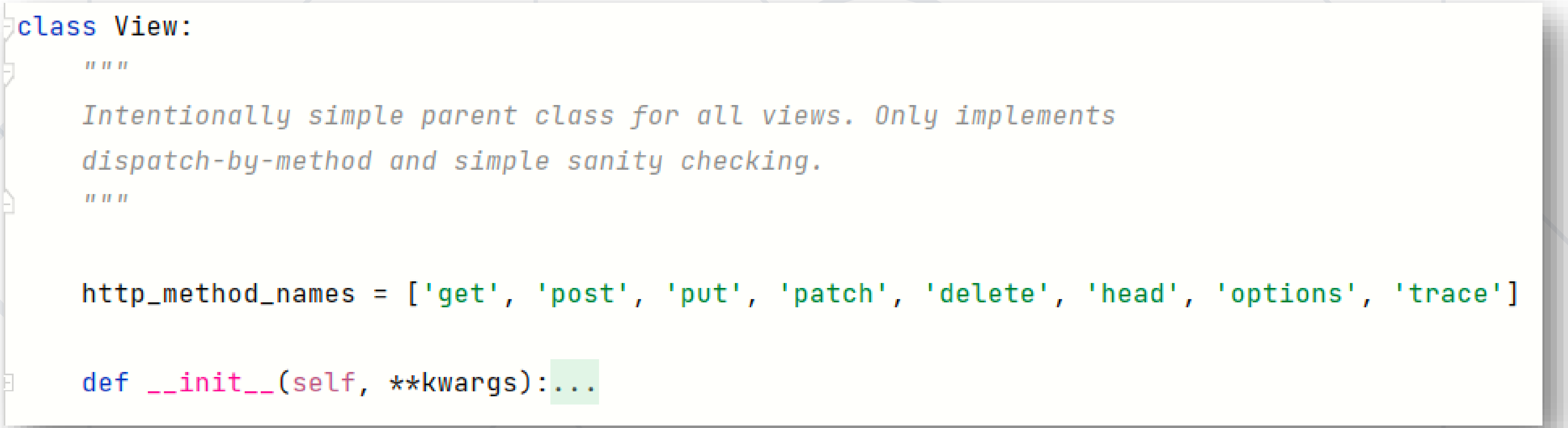
Base Views

Base Views

- "**Parent**" views, which can be **used by themselves** or **inherited from**
- Provide much of the **functionality needed** to create Django views
  - However, they may **not provide all the capabilities** required for projects
- They are positioned in the **base.py** module

The View Class

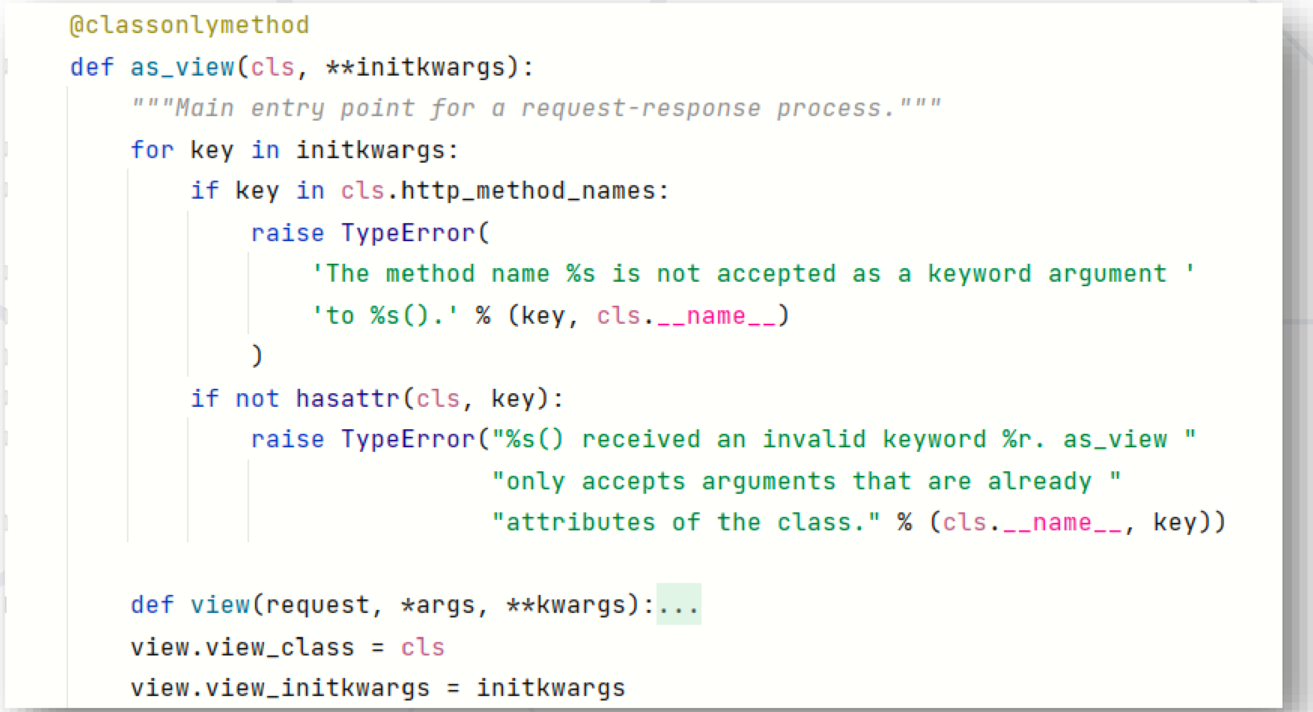
- The **master class-based** base view
- All other class-based views **inherit from it**
- HTTP method names that this view accept
  - **['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']**



```
class View:
    """
    Intentionally simple parent class for all views. Only implements
    dispatch-by-method and simple sanity checking.
    """

    http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']

    def __init__(self, **kwargs):...
```

The as_view method

- It is decorated by a **@classonlymethod**
  - Meaning it is only available on the **class** and **not** on an **instance**
  - It iterates over **initkwargs** and makes **validations**

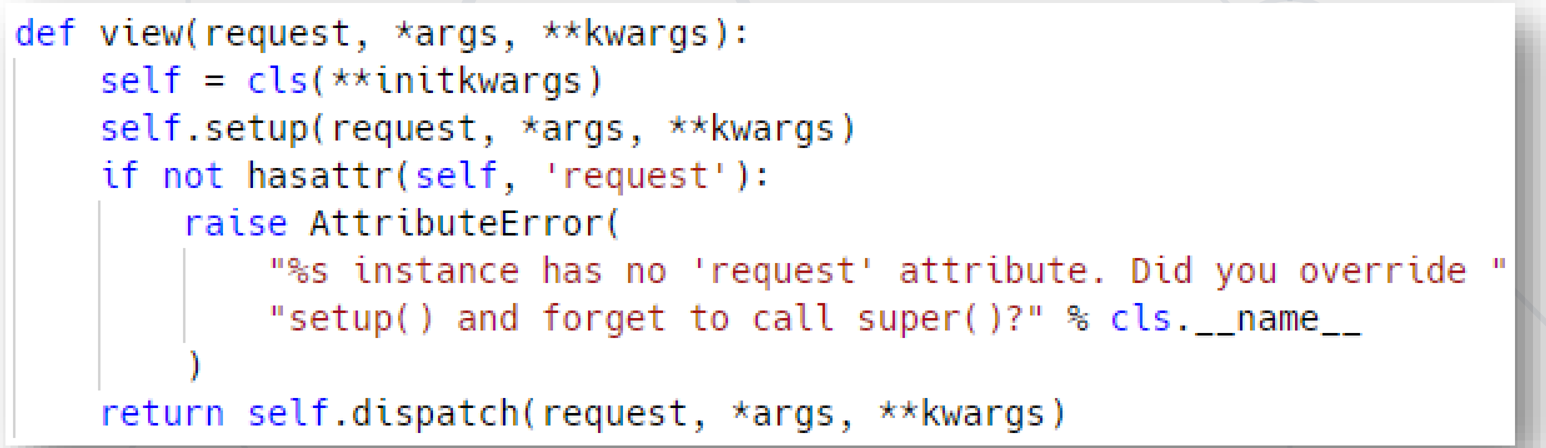


```
@classonlymethod
def as_view(cls, **initkwargs):
    """Main entry point for a request-response process."""
    for key in initkwargs:
        if key in cls.http_method_names:
            raise TypeError(
                'The method name %s is not accepted as a keyword argument '
                'to %s().' % (key, cls.__name__)
            )
        if not hasattr(cls, key):
            raise TypeError("%s() received an invalid keyword %r. as_view "
                            "only accepts arguments that are already "
                            "attributes of the class." % (cls.__name__, key))

    def view(request, *args, **kwargs):...
    view.view_class = cls
    view.view_initkwargs = initkwargs
```

The `view` method

- It accepts **request**, ***args**, ****kwargs**
- It binds
  - self to the class attributes **initkwargs
  - self.request = request
  - self.args = args
  - self.kwargs = kwargs

Creating a function that wraps around an instance of our class, and executes `dispatch()` on that instance

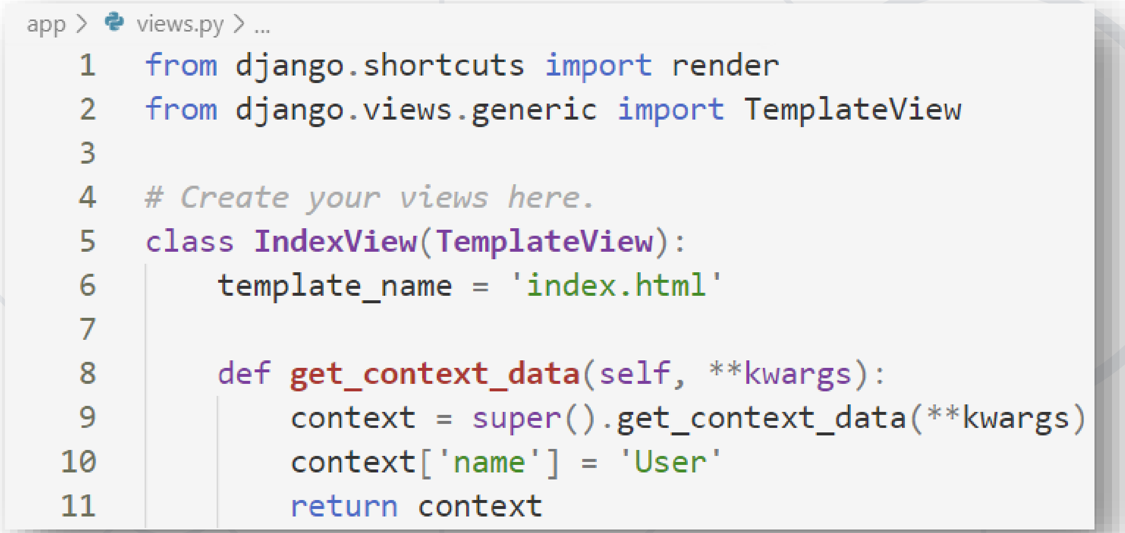


```python
def view(request, *args, **kwargs):
    self = cls(**initkwargs)
    self.setup(request, *args, **kwargs)
    if not hasattr(self, 'request'):
        raise AttributeError(
            "%s instance has no 'request' attribute. Did you override "
            "setup() and forget to call super()?" % cls.__name__
        )
    return self.dispatch(request, *args, **kwargs)
```

The TemplateView Class

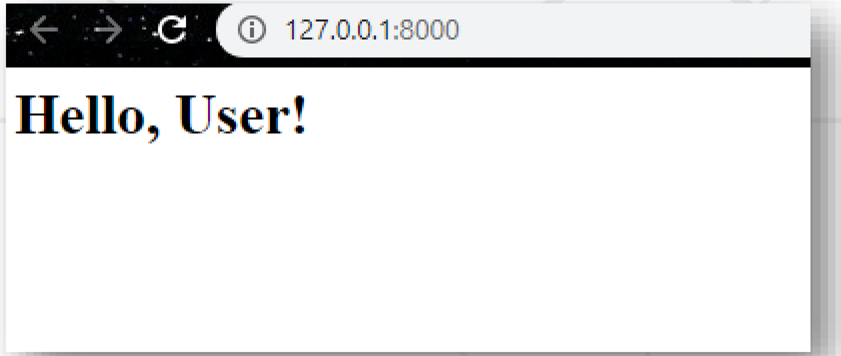
- A template view **renders** a given **template**, with the **context** containing parameters captured in the URL
- It **inherits** methods and attributes from the following
  - **`TemplateResponseMixin`**
  - **`ContextMixin`**
  - **`View`**

Basic Template View Example



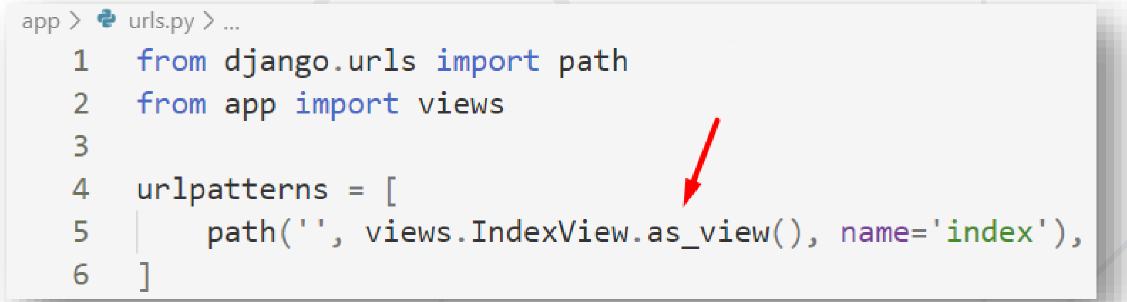
app > views.py > ...

```python
from django.shortcuts import render
from django.views.generic import TemplateView

# Create your views here.
class IndexView(TemplateView):
    template_name = 'index.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['name'] = 'User'
        return context
```



127.0.0.1:8000

Hello, User!

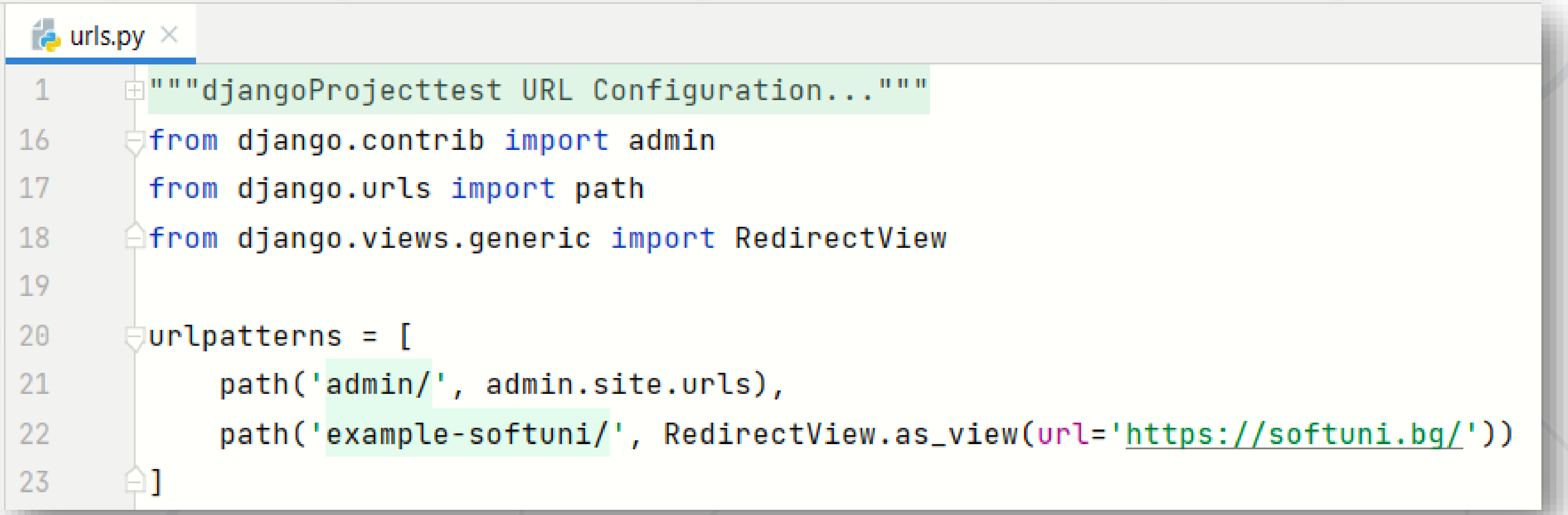


app > urls.py > ...

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

urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
]
```

The RedirectView Class

- Redirects to a given URL
- It **inherits** from the **`View`** class only



urls.py

```python
"""djangoProjecttest URL Configuration..."""
from django.contrib import admin
from django.urls import path
from django.views.generic import RedirectView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('example-softuni/', RedirectView.as_view(url='https://softuni.bg/'))
]
```

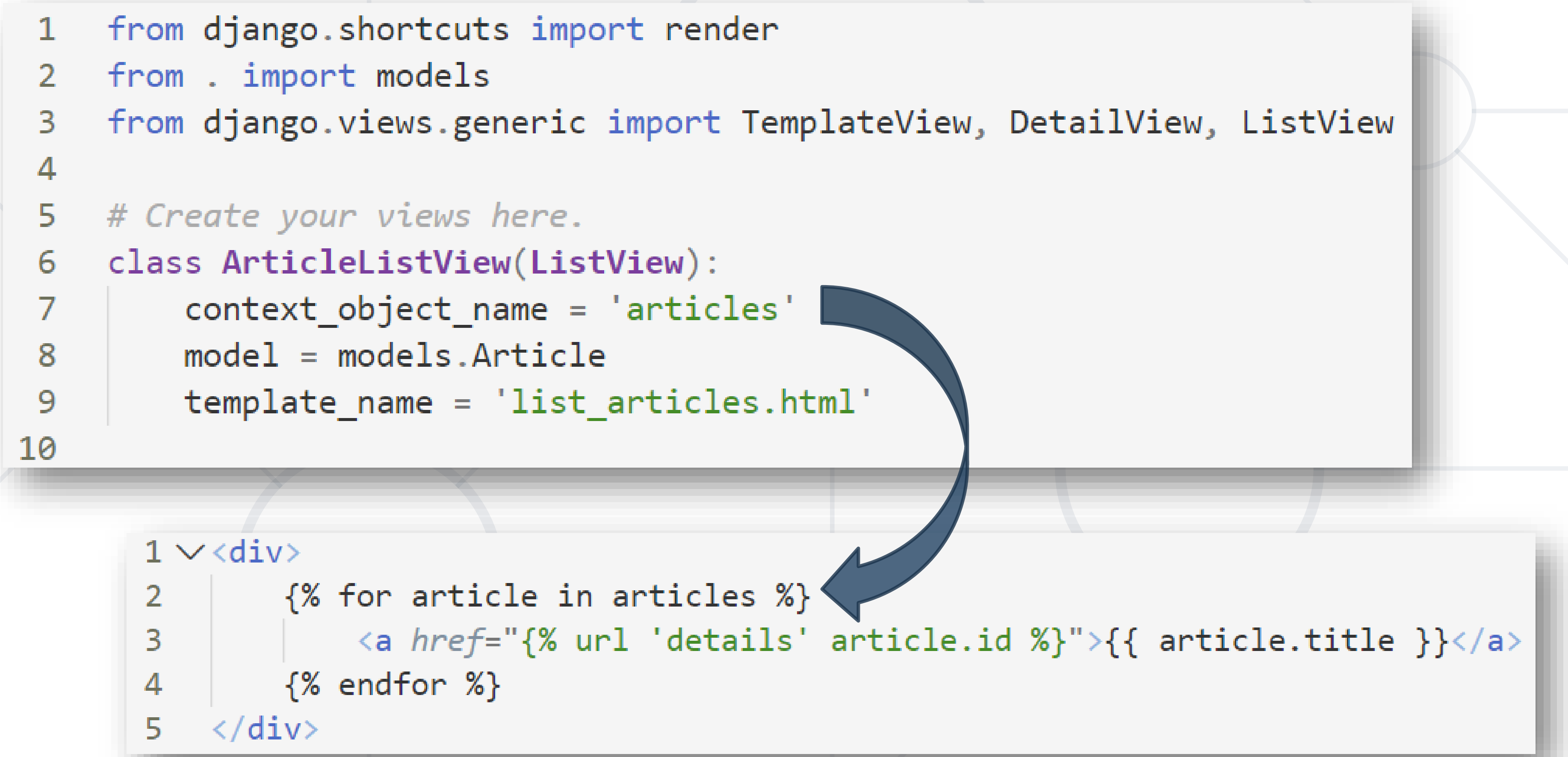
Generic Views

Built-in Generic Views

- Ease the monotonous development process
  - Provide interfaces to perform the **most common tasks** developers encounter
- Generic views:
  - Display **list and detail pages** for a single object
  - Allow users to **create**, **update**, and **delete** objects
  - Present date-based objects in **year**/**month**/**day archive** pages

Basic List View Example

- A list view is used for representing a **list of objects**

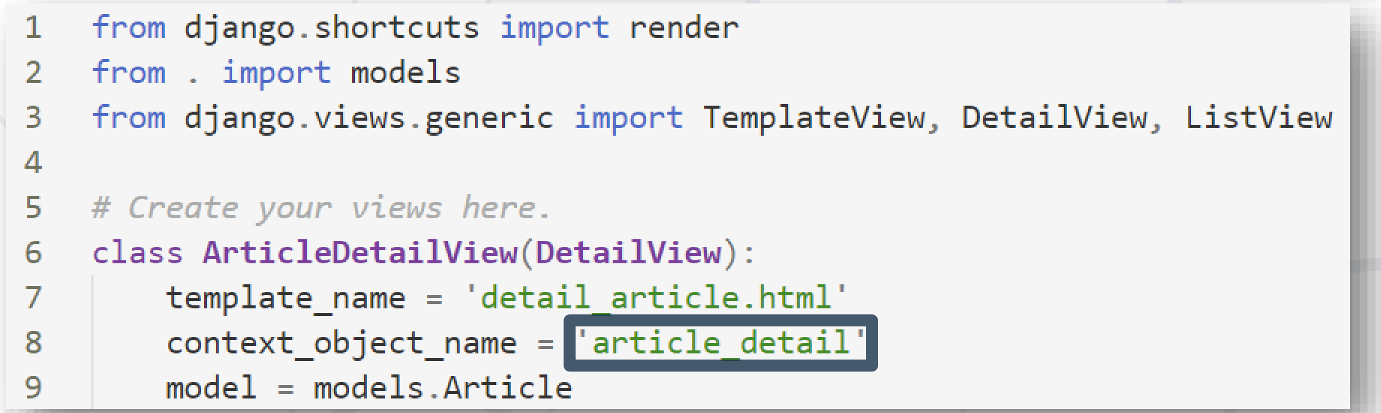


```python
from django.shortcuts import render
from . import models
from django.views.generic import TemplateView, DetailView, ListView

# Create your views here.
class ArticleListView(ListView):
    context_object_name = 'articles'
    model = models.Article
    template_name = 'list_articles.html'

```

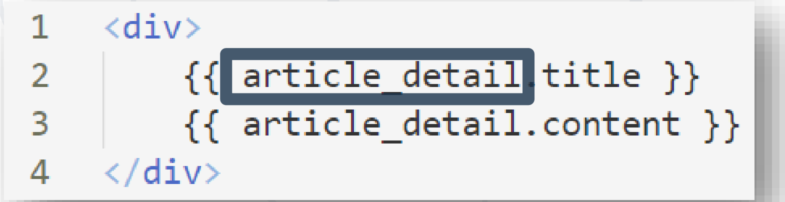
```html
<div>
    {% for article in articles %}
        <a href="{% url 'details' article.id %}">{{ article.title }}</a>
    {% endfor %}
</div>
```

Basic Detail View Example

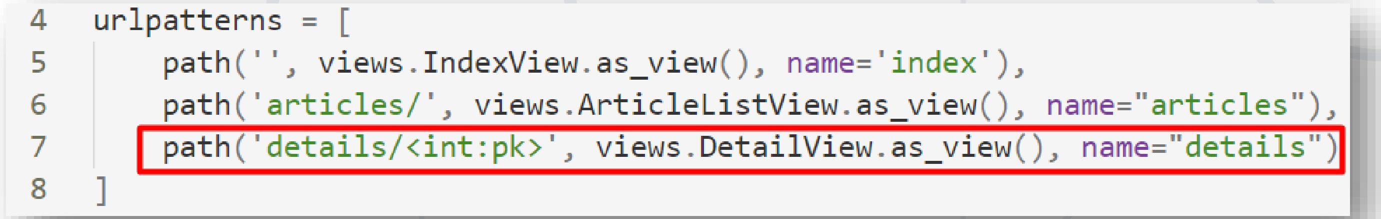
- While this view is executing, `self.object` will contain the object that the view is operating upon



```python
from django.shortcuts import render
from . import models
from django.views.generic import TemplateView, DetailView, ListView

# Create your views here.
class ArticleDetailView(DetailView):
    template_name = 'detail_article.html'
    context_object_name = 'article_detail'
    model = models.Article
```



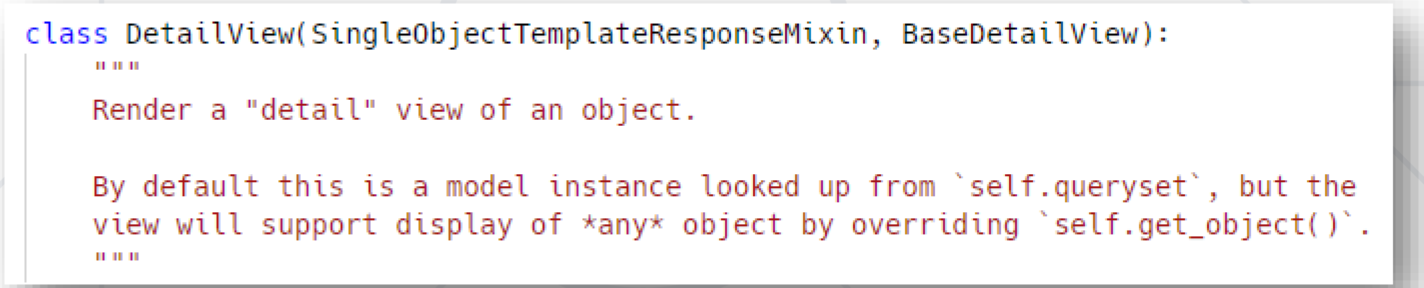
```html
<div>
    {{ article_detail.title }}
    {{ article_detail.content }}
</div>
```



```python
urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('articles/', views.ArticleListView.as_view(), name="articles"),
    path('details/<int:pk>', views.DetailView.as_view(), name="details")
]
```

DetailView inheritance structure

- The DetailView is defined in **django/views/generic/details.py** file

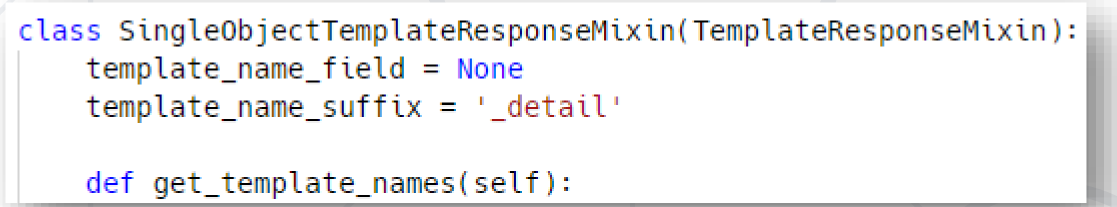


```
class DetailView(SingleObjectTemplateResponseMixin, BaseDetailView):
    """
    Render a "detail" view of an object.

    By default this is a model instance looked up from `self.queryset`, but the
    view will support display of *any* object by overriding `self.get_object()`.
    """
```

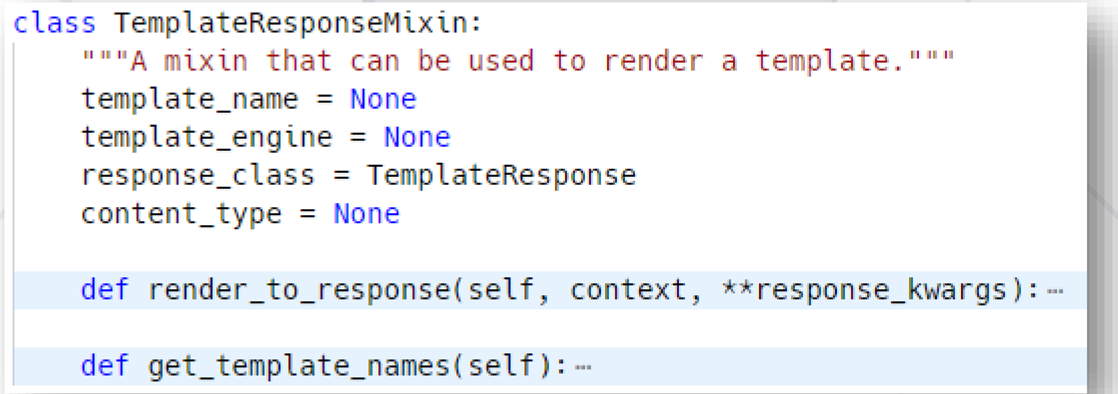
- We see here that DetailView **doesn't define** anything
- It **inherits** from **SingleObjectTemplateResponseMixin** and **BaseDetailView**

CBV's inheritance structure

- Scrolling up in the same file, we can inspect the **SingleObjectTemplateResponseMixin**
- It inherits from **TempleteResponseMixin**



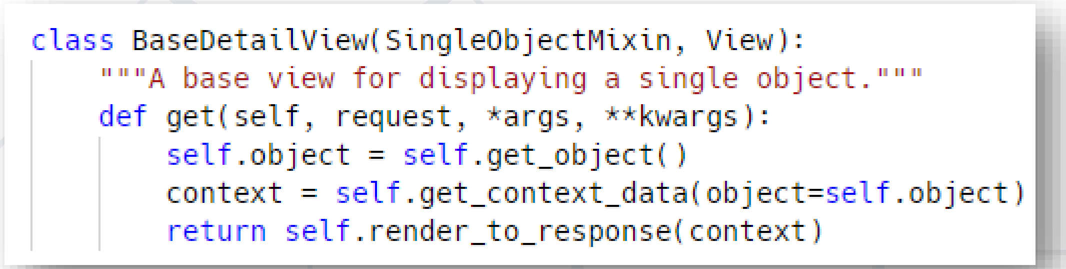
```python
class SingleObjectTemplateResponseMixin(TemplateResponseMixin):
    template_name_field = None
    template_name_suffix = '_detail'

    def get_template_names(self):
```



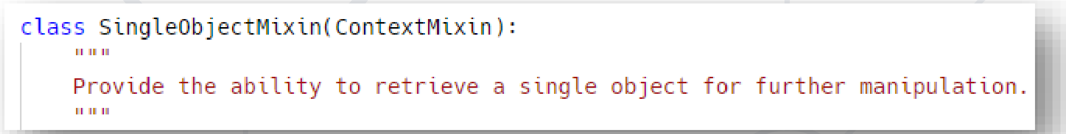
```python
class TemplateResponseMixin:
    """A mixin that can be used to render a template."""
    template_name = None
    template_engine = None
    response_class = TemplateResponse
    content_type = None

    def render_to_response(self, context, **response_kwargs):...

    def get_template_names(self):...
```

CBV's inheritance structure

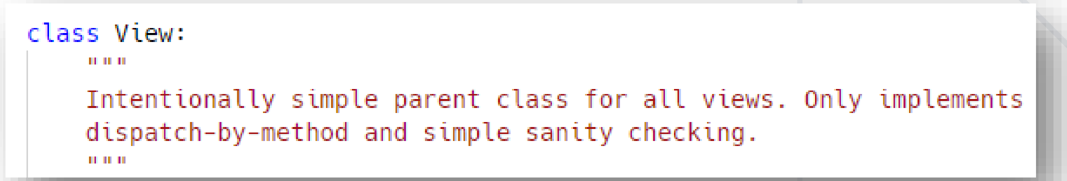
- Going a step back, it is now time to check out the **BaseDetailView**, and it inherits from two things
  - **SingleObjectMixin**
  - **View**



```python
class BaseDetailView(SingleObjectMixin, View):
    """A base view for displaying a single object."""
    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        return self.render_to_response(context)
```



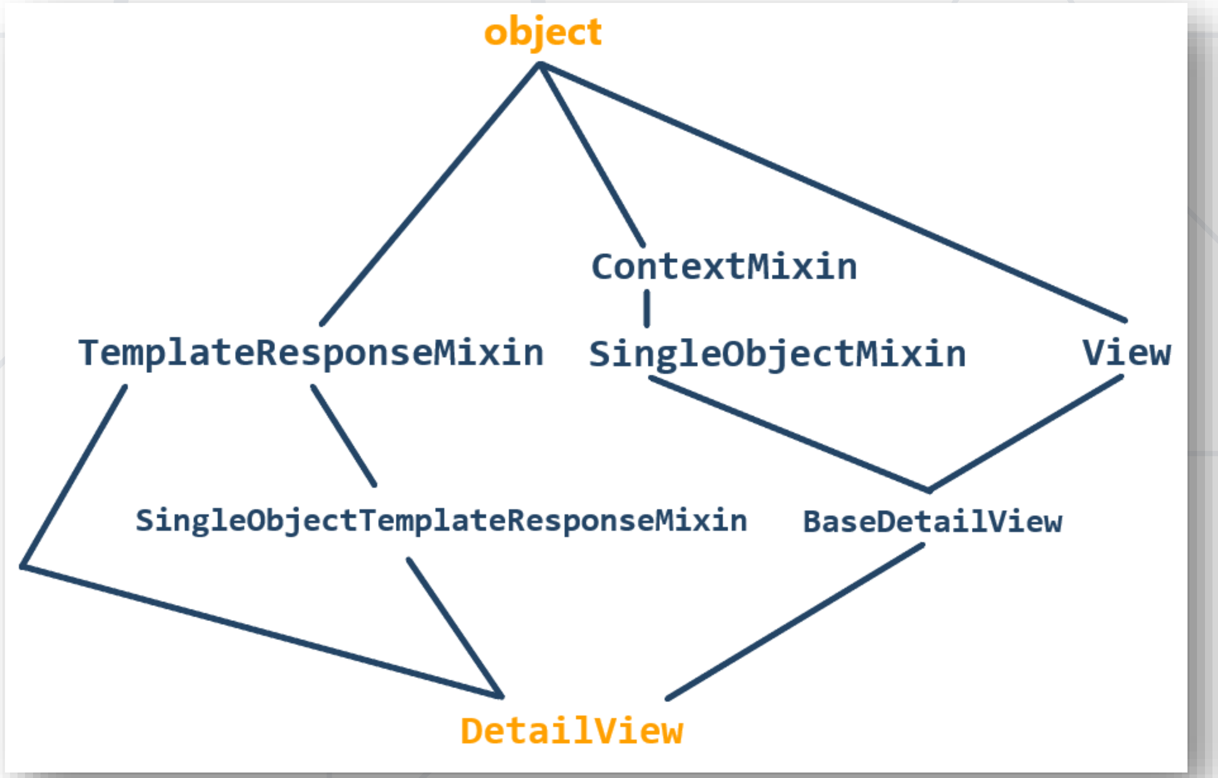
```python
class SingleObjectMixin(ContextMixin):
    """
    Provide the ability to retrieve a single object for further manipulation.
    """
```



```python
class View:
    """
    Intentionally simple parent class for all views. Only implements
    dispatch-by-method and simple sanity checking.
    """
```

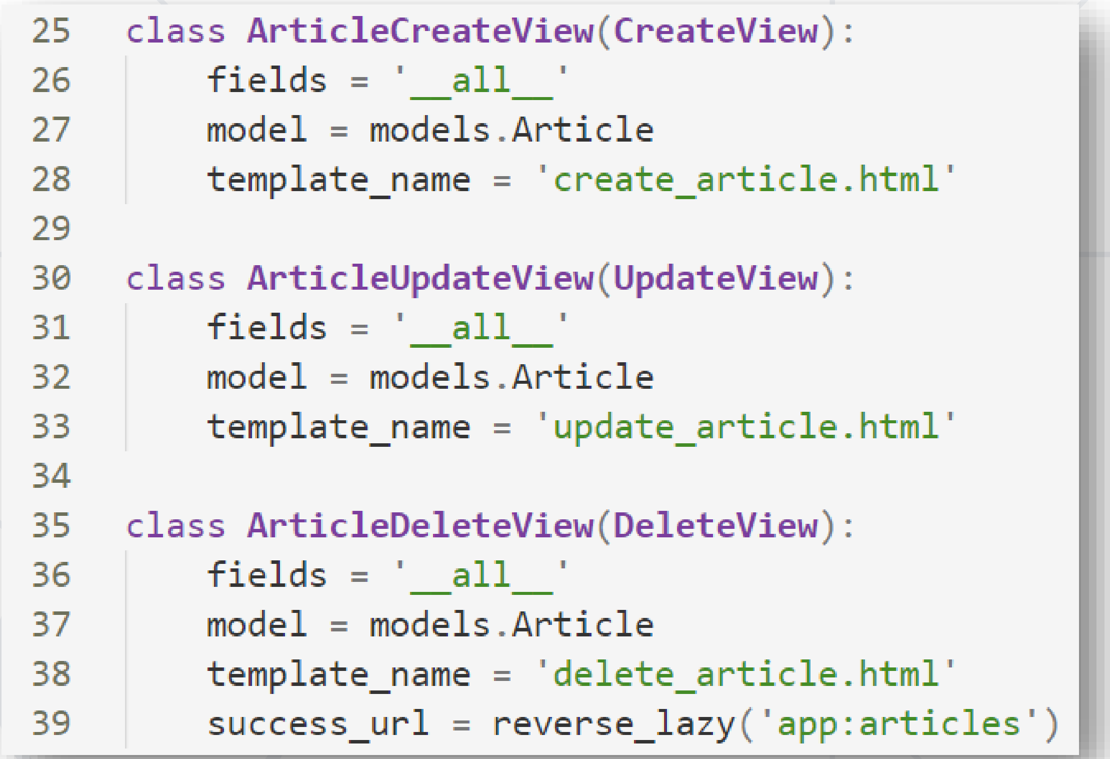
CBV's inheritance structure

- Finally, we find that **ContextMixin**, **TemplateResponseMixin** and **View** all inherit from **object**



CRUD Views

- A **C**reate view displays a form for creating an object
- An **U**pdate view displays a form for editing an existing object
- A **D**elete view displays a confirmation page and deletes an existing object

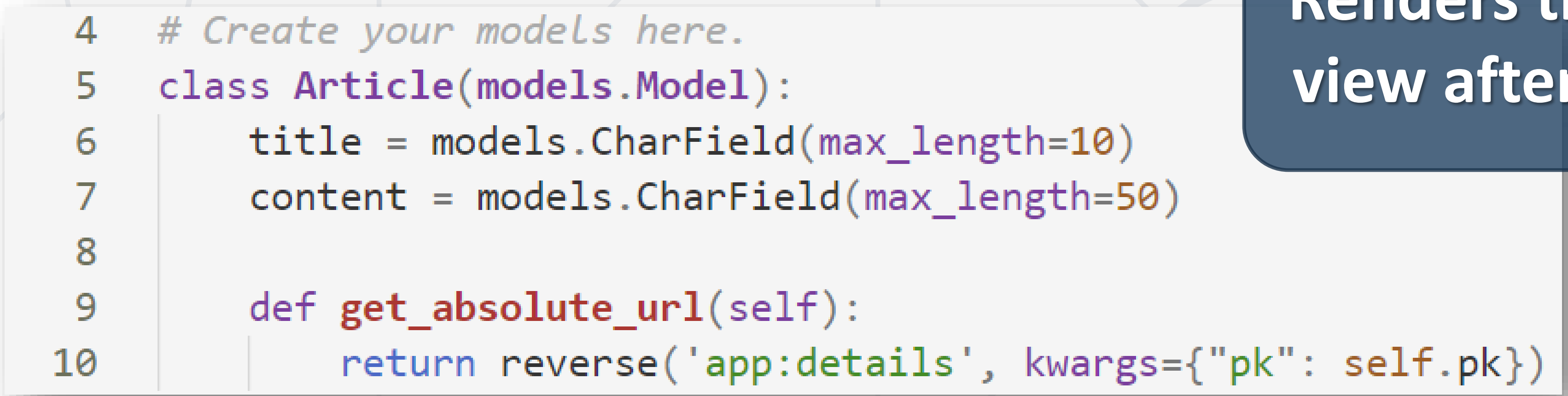


```python
class ArticleCreateView(CreateView):
    fields = '__all__'
    model = models.Article
    template_name = 'create_article.html'

class ArticleUpdateView(UpdateView):
    fields = '__all__'
    model = models.Article
    template_name = 'update_article.html'

class ArticleDeleteView(DeleteView):
    fields = '__all__'
    model = models.Article
    template_name = 'delete_article.html'
    success_url = reverse_lazy('app:articles')
```

Action on success

Set up absolute URL

- When using a **CreateView**, we need to use a function in the model called **get_absolute_url()**
- We use it to tell Django how to calculate the **canonical URL** for an object

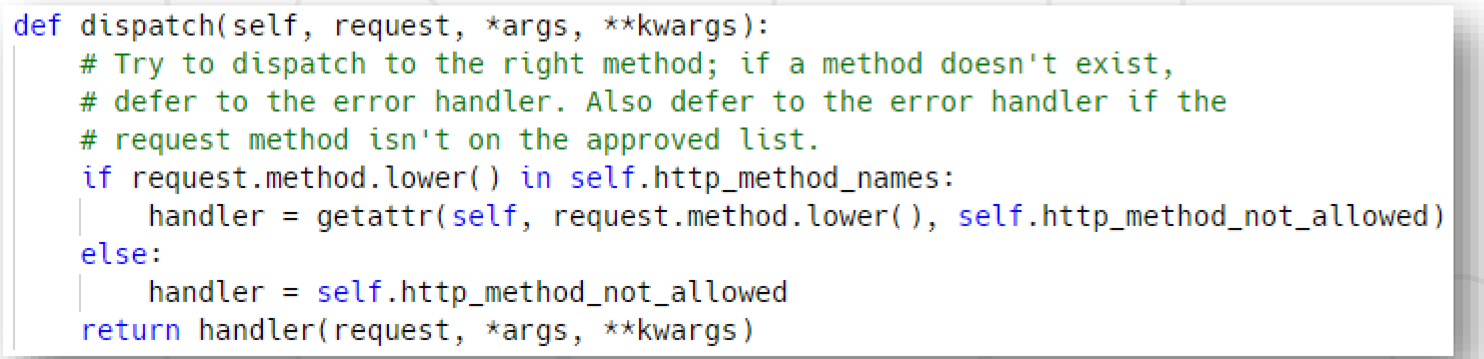


```python
# Create your models here.
class Article(models.Model):
    title = models.CharField(max_length=10)
    content = models.CharField(max_length=50)

    def get_absolute_url(self):
        return reverse('app:details', kwargs={"pk": self.pk})
```

Renders the details view after creation

Useful CBVs Methods

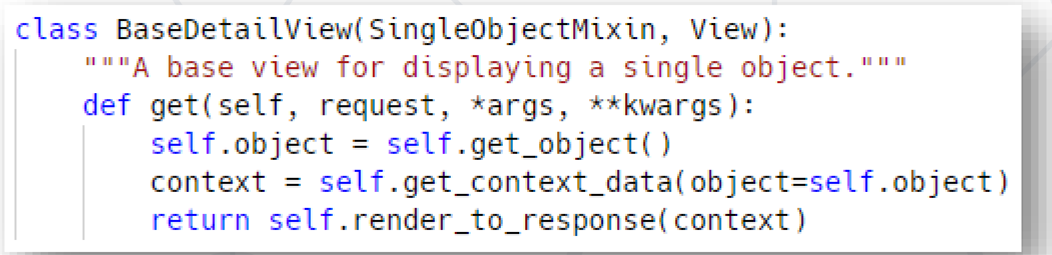
CBV dispatch()



```python
def dispatch(self, request, *args, **kwargs):
    # Try to dispatch to the right method; if a method doesn't exist,
    # defer to the error handler. Also defer to the error handler if the
    # request method isn't on the approved list.
    if request.method.lower() in self.http_method_names:
        handler = getattr(self, request.method.lower(), self.http_method_not_allowed)
    else:
        handler = self.http_method_not_allowed
    return handler(request, *args, **kwargs)
```

- The view part of the view – the method that accepts a **request**, ***args**, ****kwargs**, and returns an **HTTP response**
- It inspects the HTTP method and attempts to delegate to a method that matches the HTTP method

CBV dispatch()

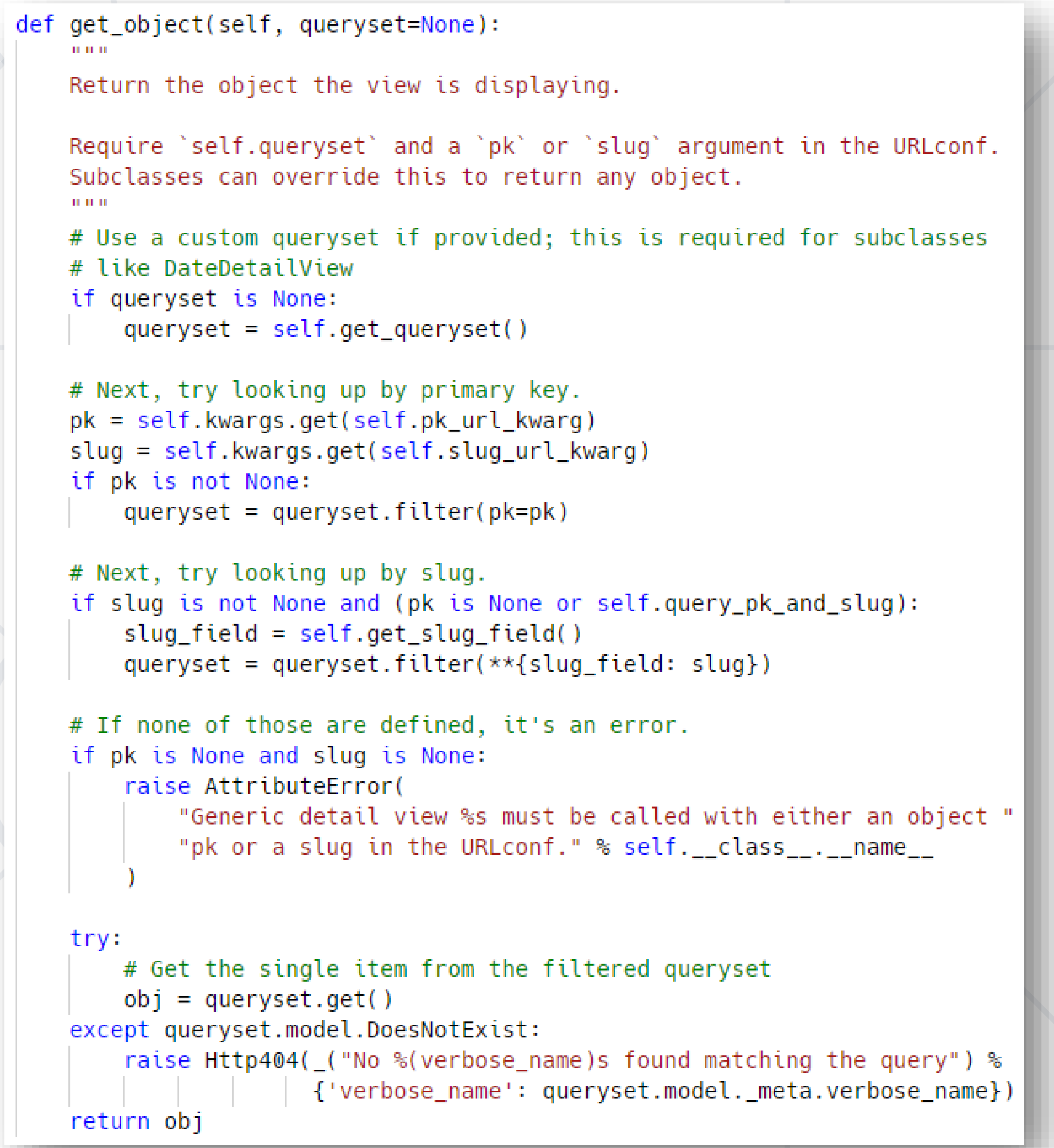


```
class BaseDetailView(SingleObjectMixin, View):
    """A base view for displaying a single object."""
    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        return self.render_to_response(context)
```

- get() accepts the **`request`**, **`*args`**, **`**kwargs`**
- It binds **`self.object`** to **`self.get_object`**
- It binds **`context`** to **`self.get_context_data`**
- It returns **`self.render_to_response(context)`**
- The **`get_object()`** method is found in the **`SingleObjectMixin`** class

The get_object method

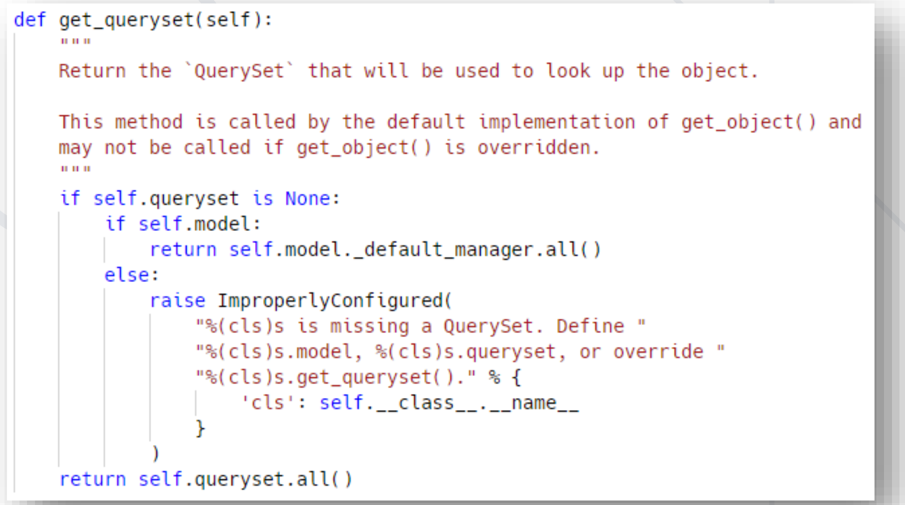
- It is method from the **SingleObjectMixin** class
- Returns a **single object** that the view will display
  - If **queryset is provided**, that queryset will be used as the source of objects
  - Performs a primary-key based lookup using the **pk argument from the URL path**



```python
def get_object(self, queryset=None):
    """
    Return the object the view is displaying.

    Require `self.queryset` and a `pk` or `slug` argument in the URLconf.
    Subclasses can override this to return any object.
    """
    # Use a custom queryset if provided; this is required for subclasses
    # like DateDetailView
    if queryset is None:
        queryset = self.get_queryset()

    # Next, try looking up by primary key.
    pk = self.kwargs.get(self.pk_url_kwarg)
    slug = self.kwargs.get(self.slug_url_kwarg)
    if pk is not None:
        queryset = queryset.filter(pk=pk)

    # Next, try looking up by slug.
    if slug is not None and (pk is None or self.query_pk_and_slug):
        slug_field = self.get_slug_field()
        queryset = queryset.filter(**{slug_field: slug})

    # If none of those are defined, it's an error.
    if pk is None and slug is None:
        raise AttributeError(
            "Generic detail view %s must be called with either an object "
            "pk or a slug in the URLconf." % self.__class__.__name__
        )

    try:
        # Get the single item from the filtered queryset
        obj = queryset.get()
    except queryset.model.DoesNotExist:
        raise Http404(_("No %(verbose_name)s found matching the query") %
                      {'verbose_name': queryset.model._meta.verbose_name})
    return obj
```

The get_queryset method

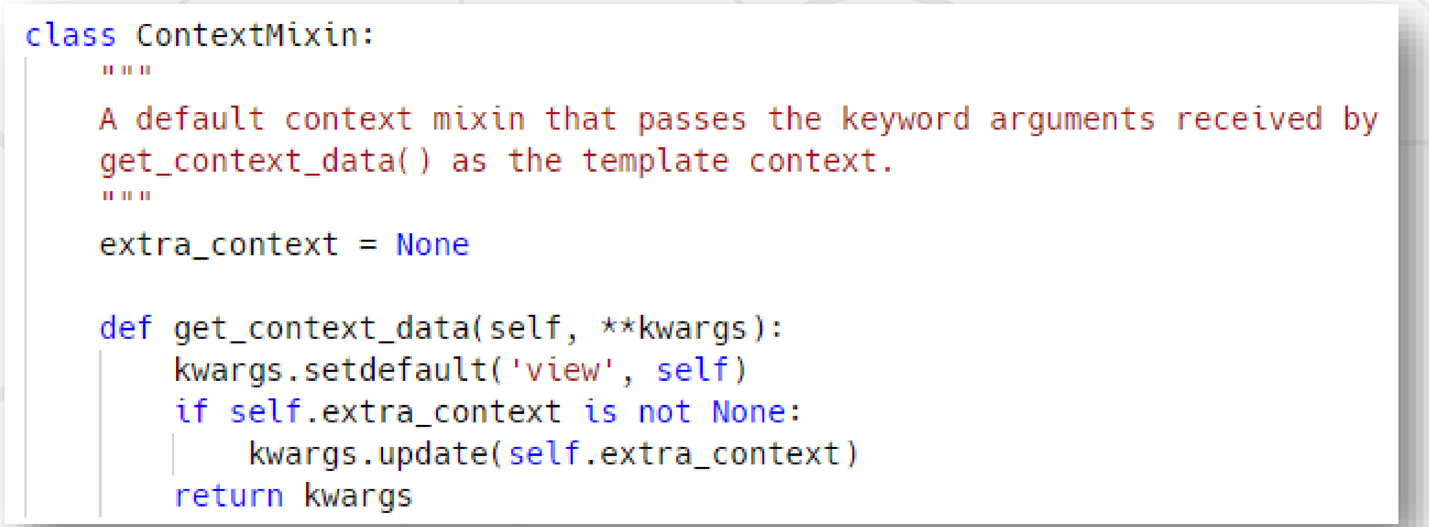
- Returns the queryset that will be used to **retrieve the object** that the view will display
- If it is not set, it **constructs a QuerySet** by calling the `all()` method on the model attribute's default manager
- Otherwise, if you don't have a model or queryset then an `ImproperlyConfigured` error is thrown that says "we have no idea what you are looking for"



```python
def get_queryset(self):
    """
    Return the `QuerySet` that will be used to look up the object.

    This method is called by the default implementation of get_object() and
    may not be called if get_object() is overridden.
    """
    if self.queryset is None:
        if self.model:
            return self.model._default_manager.all()
        else:
            raise ImproperlyConfigured(
                "%(cls)s is missing a QuerySet. Define "
                "%(cls)s.model, %(cls)s.queryset, or override "
                "%(cls)s.get_queryset()." % {
                    'cls': self.__class__.__name__
                }
            )
    return self.queryset.all()
```

The get_context_data()

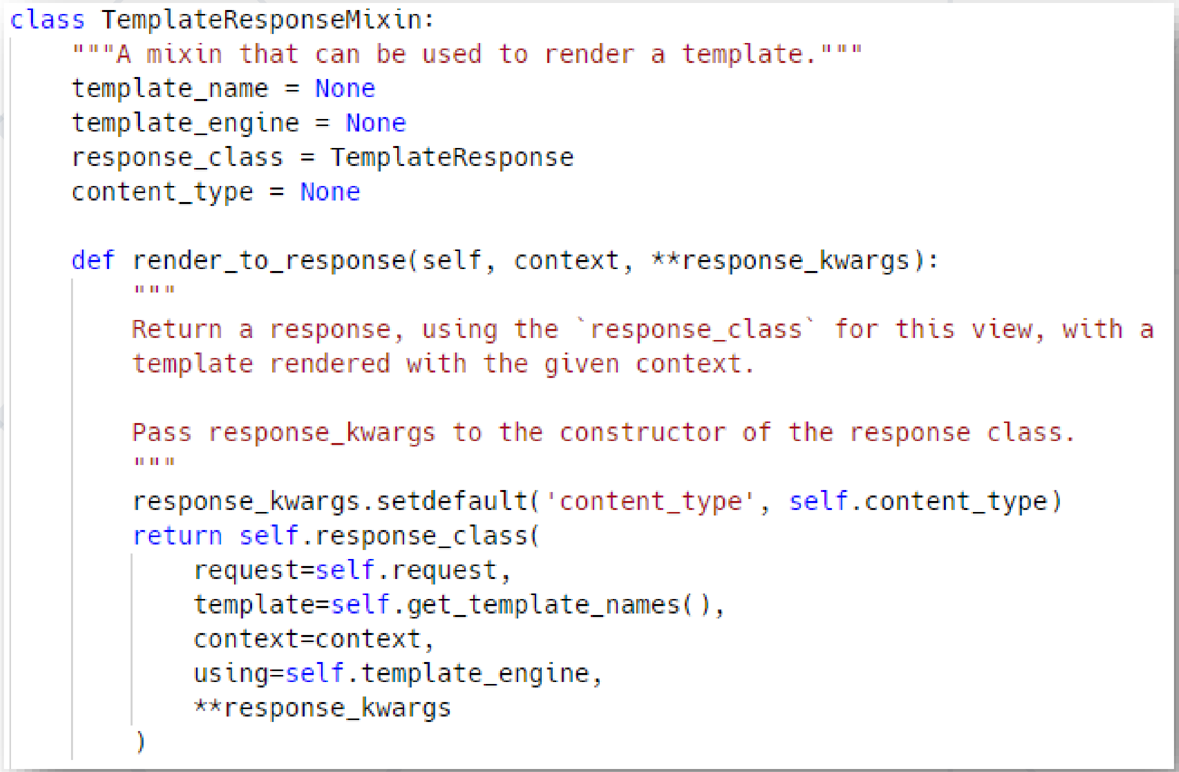
- Returns a **dictionary** representing the template context
- The keyword arguments provided will make up the returned context
- When overriding this method, you should call the **super** method for the **`.get_context_data(**kwargs)`**



```python
class ContextMixin:
    """
    A default context mixin that passes the keyword arguments received by
    get_context_data() as the template context.
    """
    extra_context = None

    def get_context_data(self, **kwargs):
        kwargs.setdefault('view', self)
        if self.extra_context is not None:
            kwargs.update(self.extra_context)
        return kwargs
```

The render_to_response()

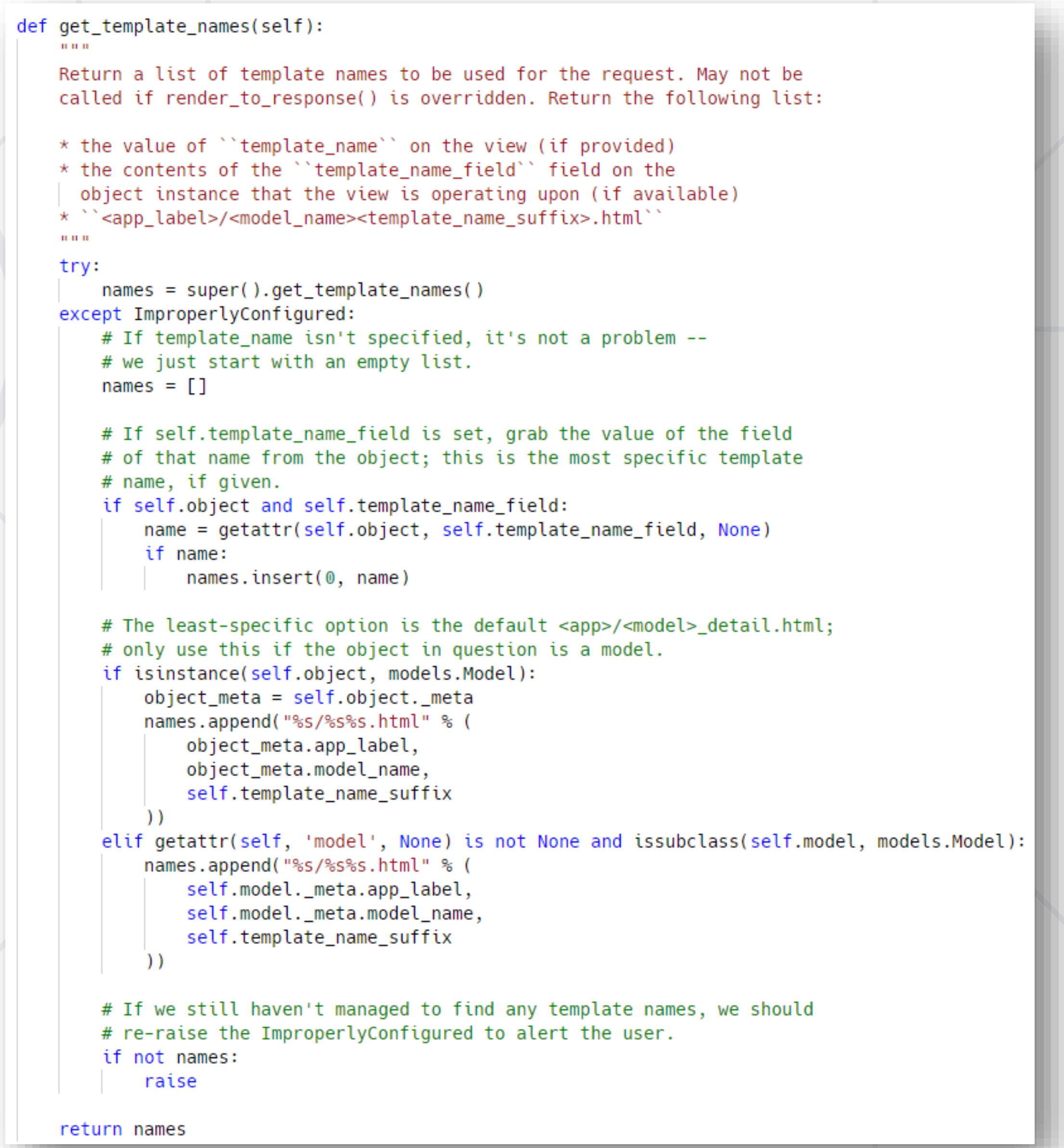
- Returns a **self.response_class** instance
- If any keyword arguments are provided, they will be **passed to the constructor** of the response class
- Calls **get_template_names()** to obtain the list of template names that will be searched looking for an existent template



```python
class TemplateResponseMixin:
    """A mixin that can be used to render a template."""
    template_name = None
    template_engine = None
    response_class = TemplateResponse
    content_type = None

    def render_to_response(self, context, **response_kwargs):
        """
        Return a response, using the `response_class` for this view, with a
        template rendered with the given context.

        Pass response_kwargs to the constructor of the response class.
        """
        response_kwargs.setdefault('content_type', self.content_type)
        return self.response_class(
            request=self.request,
            template=self.get_template_names(),
            context=context,
            using=self.template_engine,
            **response_kwargs
        )
```

The get_template_names() method

- Returns a **list of template names** to search for when rendering the template
- The **first template** that is found will be used
- The default implementation will return a **list** containing **template_name** (if it is specified)



```python
def get_template_names(self):
    """
    Return a list of template names to be used for the request. May not be
    called if render_to_response() is overridden. Return the following list:

    * the value of ``template_name`` on the view (if provided)
    * the contents of the ``template_name_field`` field on the
      object instance that the view is operating upon (if available)
    * ``<app_label>/<model_name><template_name_suffix>.html``
    """
    try:
        names = super().get_template_names()
    except ImproperlyConfigured:
        # If template_name isn't specified, it's not a problem --
        # we just start with an empty list.
        names = []

        # If self.template_name_field is set, grab the value of the field
        # of that name from the object; this is the most specific template
        # name, if given.
        if self.object and self.template_name_field:
            name = getattr(self.object, self.template_name_field, None)
            if name:
                names.insert(0, name)

        # The least-specific option is the default <app>/<model>_detail.html;
        # only use this if the object in question is a model.
        if isinstance(self.object, models.Model):
            object_meta = self.object._meta
            names.append("%s/%s%s.html" % (
                object_meta.app_label,
                object_meta.model_name,
                self.template_name_suffix
            ))
        elif getattr(self, 'model', None) is not None and issubclass(self.model, models.Model):
            names.append("%s/%s%s.html" % (
                self.model._meta.app_label,
                self.model._meta.model_name,
                self.template_name_suffix
            ))

        # If we still haven't managed to find any template names, we should
        # re-raise the ImproperlyConfigured to alert the user.
        if not names:
            raise

    return names
```

Practice Time

Using Class-Based-Views

Summary

- Class-based views provide an alternative way to implement views as Python **objects** instead of **functions**
- CBV's are **easily extended**, as function views are **easier** to implement
- To practice, try **redoing** your older projects and use **CBV's** instead of function bases ones

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

CODE IT UP

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License