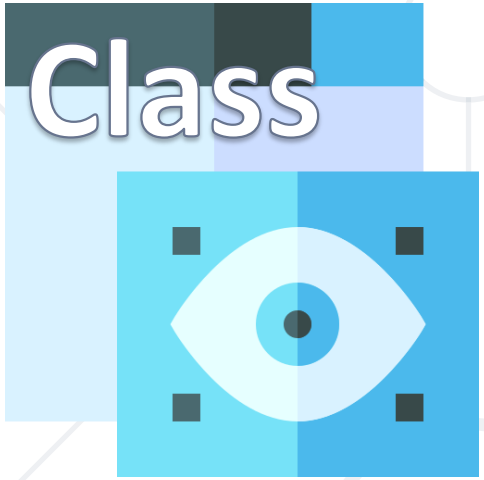
Class-Based Views Advanced



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Have a Question?

Software University

sli.do

#python-web

Table of Contents

Generic Views

Built-in Generic Views Benefits

- **Generic views** in Django are designed to
  - **simplify** the development **process**
  - by offering **convenient interfaces** for **common** tasks developers often face
- They provide a **high-level abstraction** to
  - perform **key operations**
  - **reducing** the need for **repetitive** code
  - **accelerating** development

Generic Views - Main Functionalities

- The main **functionalities** of **generic** views include
  - Displaying **List** and **Detail** Pages
    - Presenting **collections** of **objects** and their **detailed** views **without** the **need** for extensive view and template code
  - CRUD Operations (**Create**, **Update**, **Delete**)
    - Offering **predefined** **views** for creating, updating, and deleting **objects**
  - **Date-Based** Object Presentation
    - A straightforward **solution** for presenting the **objects** in **year**/**month**/**day** **archive** pages

Basic DetailView

```
class ArticleDetailView(DetailView):
    model = Article
    template_name = 'article_details.html'
    context_object_name = 'article'
```

context_object_name value replaces the default value - object

```
urlpatterns = [
    ...
    path('article/<int:pk>/', ArticleDetailView.as_view(),
name='details'),
]
```

```
<div>
        <h1>{{ article.title }}</h1>
        <p>{{ article.content }}</p>
</div>
```

Basic DetailView

- In a Django **DetailView**, the **self.object** attribute **refers** to the **object** that the view is **operating** upon
- This **object** is typically an **instance** of the model specified in the **model** attribute and is **set** by the **SingleObjectMixin** during the view **processing lifecycle**

```
class ArticleDetailView(DetailView):
    model = Article
    template_name = 'article_details.html'

    def get(self, request, *args, **kwargs):
        # Access the current object being viewed
        current_article = self.get_object()
        # You can do something with the current_article
        return super().get(request, *args, **kwargs)
```

SingleObjectMixin method

get_object() Method

Software University

- The primary purpose of the `get_object` method is to
  - **fetch** and **return** a **single object**
  - based on the view's **configuration** and the **URL parameters**
- It is particularly useful in **detail views** where you need to
  - **display** information about a **specific instance** of a **model**
- If the method is **not overridden** in a class-based view
  - The **default behavior** is provided by the `SingleObjectMixin` class
  - The **default implementation** uses the **model** specified in the **model attribute** and the **primary key** from the **URL parameters** to **retrieve** the corresponding **object**

get_object() Method

Software University

```
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
```

get_object() Method - Example

- By understanding and utilizing the **get_object** method
  - Developers can have **greater control** over how **individual objects** are **retrieved** and **processed** in their class-based views

```
class ArticleDetailView(DetailView):
    ...

    def get_object(self, queryset=None):
        # Retrieve the current object
        current_object = super().get_object(queryset)
        # Update view count for the article
        current_object.views_count += 1
        current_object.save()
        return current_object
```

DetailView Inheritance Chain

Software University

- The **DetailView** class is defined in **`django/views/generic/detail.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()`.
    """
```

- In this file, we observe that the **DetailView** itself does **not define** any **methods** or **attributes** specific to its behavior
  - Instead, it **relies** on the **functionality** provided by its **parent classes**
  - **`SingleObjectTemplateResponseMixin`** and **`BaseDetailView`**

DetailView Inheritance Chain

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

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

    def get_template_names(self):
```

```
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):…
```

DetailView Inheritance Chain

- Going a step back, let's explore the **`BaseDetailView`** and its immediate **parent classes**
  - **`SingleObjectMixin`** and **`View`**

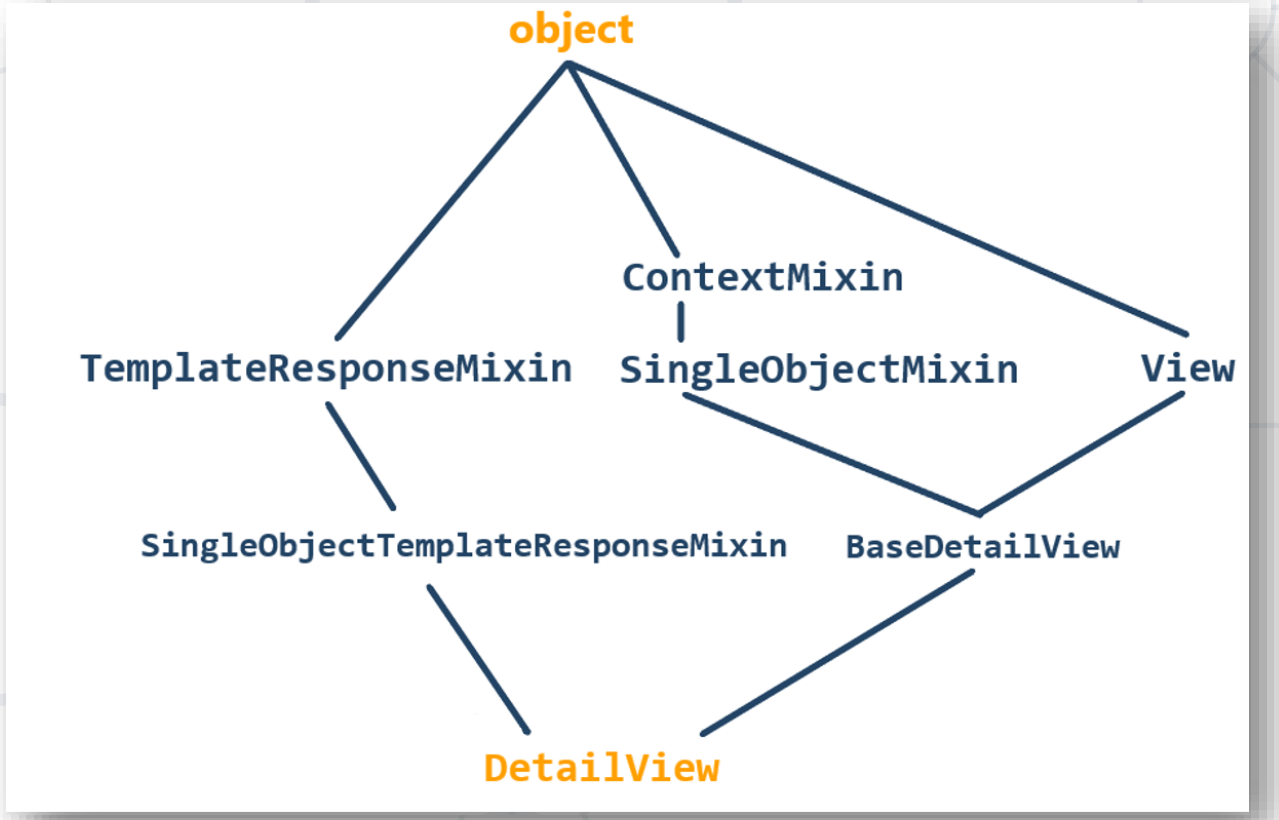
```python
class BaseDetailView(SingleObjectMixin, View):
    """A base view for displaying a single object."""
    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        return self.render_to_response(context)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

```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.
    """
```

DetailView Inheritance Chain

- Continuing the exploration, we observe that `ContextMixin`, `TemplateResponseMixin`, and `View` all inherit from the base Python class `object`



get_absolute_url() Method

- When using a **DetailView**, it is **beneficial** to **define** a method in the **model** called `get_absolute_url()`
- This method is used to **inform** Django how to **calculate** the **canonical URL** for an **object**
- By implementing `get_absolute_url()`, you provide a **standardized** way for Django to **determine** the URL **associated** with a **particular instance** of the model

```
class Article(models.Model):
    ...

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

URL pattern name

Generates the canonical URL for an instance of the Article model

Basic ListView

- The **ListView** is specifically **designed** to display 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'

```

context_object_name value replaces the default value - **object_list**

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

ListView Inheritance Chain

- Similar to **DetailView**, the **ListView** inherits from two main classes:
  - `MultipleObjectTemplateResponseMixin` and `BaseListView`
- The **ListView** also does **not define** any **methods** or **attributes** specific to its behavior
  - Instead, it **relies** on the **functionality** provided by its **parent** **classes**

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
    """
    Render some list of objects, set by `self.model` or `self.queryset`.
    `self.queryset` can actually be any iterable of items, not just a queryset.
    """
```

get_queryset() Method

Software University

- **Returns** the **QuerySet** that will be **used** to **retrieve** the **objects** the view will display
- The **default** behavior is to **construct** a **QuerySet** by **calling** the **`all()` method** on the model's **default** manager
- If **neither** a model **nor** a queryset is provided, the **absence** of this crucial information leads to an **`ImproperlyConfigured`** error
  - The system **cannot determine** what **data** to **retrieve** and **display**
  - Prompting developers to **address** the **missing** configuration

get_queryset() Method

```
def get_queryset(self):
    """
    Return the list of items for this view.

    The return value must be an iterable and may be an instance of
    `QuerySet` in which case `QuerySet` specific behavior will be enabled.
    """
    if self.queryset is not None:
        queryset = self.queryset
        if isinstance(queryset, QuerySet):
            queryset = queryset.all()
    elif self.model is not None:
        queryset = 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__}
        )
    ordering = self.get_ordering()
    if ordering:
        if isinstance(ordering, str):
            ordering = (ordering,)
        queryset = queryset.order_by(*ordering)

    return queryset
```

Pagination

- **Pagination** is the process of **dividing** a **large set** of data into **smaller**, more manageable **chunks** called "**pages**"
- This helps **improve** the **user experience**
  - By displaying a **limited number** of **items** per **page**
  - Making it **easier** to **navigate** through **extensive** datasets
- Django provides a **built-in** **support** for **pagination** through
  - `Paginator` and `Page` classes

Pagination - Example

```
# views.py
from django.views.generic import ListView
from .models import Article

class ArticleListView(ListView):
    model = Article
    template_name = 'list_articles.html'
    context_object_name = 'articles'
    paginate_by = 4  # Number of articles per page

    def get_queryset(self):
        return Article.objects.all().order_by('-date_published')
        # Example queryset ordering by date_published
```

Pagination - Example

```
<!-- list_articles.html -->
{% for article in articles %}
    <!-- Display article content here -->
{% endfor %}

<div class="pagination">
    <span class="step-links">
        {% if page_obj.has_previous %}
            <a href="?page=1">&laquo; first</a>
            <a href="?page={{ page_obj.previous_page_number }}">previous</a>
        {% endif %}

        <span class="current">
            Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}.
        </span>

        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}">next</a>
            <a href="?page={{ page_obj.paginator.num_pages }}">last &raquo;</a>
        {% endif %}
    </span>
</div>
```

Useful CBVs Methods

Customizing CBVs

Customizing Generic Views

- Customizing **generic views** in Django involves using
  - **Methods** and **Attributes** provided by the views
  - Overriding them to tailor their **behavior** according to specific **requirements**
- Attributes that developers often adjust or modify
  - `model`
  - `template_name`
  - `context_object_name`
  - `paginate_by`

Customizing Generic Views

Software University

- **Methods** that developers often **customize** by **overriding** them
  - `get()`
  - `post()`
  - `get_queryset()`
  - `get_context_data()`
  - `dispatch()`

get_context_data()

- In Django's CBVs, the `get_context_data()` method is used to
  - provide **additional** **context data** to the **template** **beyond** what is **automatically** generated
- This method allows you to **customize** the **context** **dictionary** used in **rendering** the template

```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
```

```python
class ArticleListView(ListView):
    template_name = 'list_articles.html'
    extra_context = {'title': 'List of Articles'}  # Static context

    def get_context_data(self, **kwargs):  # Dynamic context
        context = super().get_context_data(**kwargs)
        context['featured_articles'] = Article.objects.filter(is_featured=True)
        context['recent_articles'] = Article.objects.order_by('-date_published')[:5]

        return context
```

get_template_names()

- The `get_template_names` method is **responsible** for **providing** the **template name** or a **list** of **template names** to be used by the view
- By default, it uses the `template_name` **attribute** specified in the view class
- However, if you need to **dynamically determine** the **template name** based on certain factors, you might **override** this **method**

get_template_names()

Software University

```
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
```

get_template_names() - Example

```
class ArticleDetailView(DetailView):
    ...

    def get_template_names(self):
        # Retrieve the current object
        current_object = self.get_object()

        # Use a different template name based on certain conditions
        if current_object.is_featured:
            return ['featured_article_template.html', 'details.html']
        return ['details.html']
```

For certain specific objects, you may prefer to use a different template

render_to_response()

Software University

- The `render_to_response` method is **provided** by the `TemplateResponseMixin`, allowing you to **customize** the **rendering process**
- You can **hook** into the **rendering** process at different **stages**, such as **modifying** the **context** or **response options** **before** the **actual** rendering occurs

```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
        )
```

```python
    def render_to_response(self, context, **response_kwargs):
        # Perform any additional processing before rendering the response
        # For example, you can modify the context or response_kwargs here

        return super().render_to_response(context, **response_kwargs)
```

Decorators and Mixins in CBVs

Decorators and Mixins in CBVs

- **Decorators** are commonly used in CBVs to **apply additional functionality** to view methods
- **Mixins** are used to **encapsulate reusable behavior** that can be **combined** with other classes
  - They are particularly useful when you like to **modularize** and **share common functionality** across **multiple** views

Custom Decorator - Example

```
from django.http import Http404
from functools import wraps
from .models import Article

def article_published_required(dispatch):
    @wraps(dispatch)
    def _wrapped_dispatch(self, request, *args, **kwargs):
        article_id = kwargs.get('pk') or kwargs.get('article_id')

        if article_id is not None:
            try:
                article = Article.objects.get(pk=article_id, is_published=True)
            except Article.DoesNotExist:
                raise Http404("Article does not exist or is not published.")
        else:
            raise Http404("Article ID not provided.")

        return dispatch(self, request, article, *args, **kwargs)

    return _wrapped_dispatch
```

Extracts the article id from the URL parameters

Checks if the article is published

Raises Http404 if the article is not found or is not published

Custom Decorator - Usage

```
from django.views.generic import DetailView
from .models import Article
from .custom_decorators import article_published_required

class ArticleDetailView(DetailView):
    model = Article
    template_name = 'details.html'
    context_object_name = 'article'

    @article_published_required
    def dispatch(self, request, *args, **kwargs):
        return super().dispatch(request, *args, **kwargs)
```

The custom decorator ensures that only published articles can be accessed through the ArticleDetailView

Mixins - Example

```
from django.views.generic import ListView
from .models import Article

class RecentArticleMixin:
    def get_queryset(self):
        return Article.objects.order_by('-date_published')[:5]

class ArticleListView(RecentArticleMixin, ListView):
    model = Article
    template_name = 'list_articles.html'
    context_object_name = 'articles'
```

Encapsulates logic for fetching recent articles

Combining functionalities

When to Use Decorators and Mixins

- **Decorators:**
  - Use **decorators** when you want to **apply specific functionality** to an **entire** view
  - They are often used for tasks like **authentication**, **permission checks**, **caching**, etc.
- **Mixins:**
  - Use **mixins** when you want to **modularize** and **share specific** pieces of **functionality** across **multiple** views
  - They are great for **promoting code reuse** and maintaining a **clean**, **organized** codebase

Summary

- Generic Views
  - DetailView, ListView
  - Pagination
- Useful CBV Methods
- Decorators and Mixins
  - Custom Decorators

Questions?

SoftUni

Software University | SoftUni AI | SoftUni Digital | SoftUni Creative | SoftUni Foundation | СофтУни БУДИТЕЛ | Finance Academy

SoftUni Diamond Partners

Software University

Coca-Cola HBC Bulgaria

Postbank

top EMPLOYER България Bulgaria 2025

FOR A BETTER WORLD OF WORK

SUPER HOSTING .BG

createX

DRAFT KINGS

THE CROWN IS YOURS

INDEAVR

Serving the high achievers

encorp.io

VIVACOM

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

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg