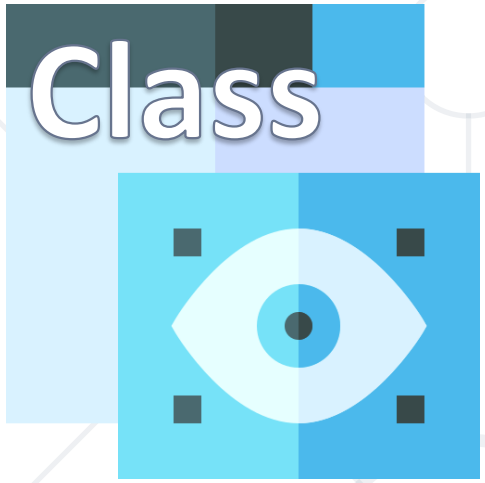
Class-Based Views Basics



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Have a Question?

Software University

sli.do

#python-web

Table of Contents

What are Class-Based Views?

What are CBVs?

- **Class-Based Views** are a programming **paradigm** in Django
  - Where **views** are defined as **Python classes** rather than functions
  - These **classes encapsulate** the **logic** for handling HTTP **requests** and **responses**

What are CBVs?

- In **Class-Based Views** (CBVs), the **self** **attribute** is used to
  - **maintain** **state** across different **methods** within the class
- Since CBVs are implemented as classes, the **instance** of the **class** (**self**) retains information
  - **throughout** the processing of a **request**

What are CBVs?

- **Class-based** views (**CBVs**) in Django offer a more **organized** and **object-oriented** approach to handling **HTTP requests**

```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')
```

CBVs Inheritance Structure

- Class-Based Views leverage class **inheritance** to create a **structured** and **modular** **organization** of **view logic**
- This approach allows
  - The **reuse** of **code**
  - The implementation of the **mixin** **pattern**

Mixin Pattern

- CBVs often employ the **mixin pattern**
- Specific functionality is **encapsulated** in **separate** classes (**mixins**)
- These **mixins** can then be **combined** by inheriting from **multiple** classes
  - providing **flexibility** and code **reuse**

CBVs vs FBVs

- **Class-Based Views**
  - Extensibility
  - Reusability and Modularity
  - Organization and Structure
  - Built-in Views
- **Function-Based Views**
  - Simplicity and Readability
  - Direct Mapping to HTTP Methods
  - Explicit Code Flow
  - More Concise for Simple Cases

CBVs vs FBVs

- **Class-Based Views**
  - Harder to Read
  - Implicit Code Flow
  - Hidden Code in Parent Classes/Mixins
- **Function-Based Views**
  - Hard to Extend
  - Hard to Reuse
  - Handling HTTP Methods via Conditional Branching

Why Using CBVs?

- **Reusability**
  - CBVs **encourage** the use of **mixins** and **inheritance**
  - Making it **easier** to **reuse** and **extend** functionality
- **Organization**
  - Views logic can be organized into **methods**, making the code more **modular** and **maintainable**
- **Built-in Views**
  - Django provides **ready-to-use** class-based views for common **patterns**

Base Views

Base Views

- **Base Views** serve as **foundational** or **parent** views in Django
- Designed to be used **independently** or **inherited** from other views
- These views offer a **substantial** **set** of **functionalities** **essential** for **crafting** Django views

Base Views

- **Base Views** provide a solid **starting point**
- It's important to note that they might **not encompass** all the **capabilities** needed for every project
- In Django, you commonly find **built-in views**, such as **generic class-based views**, in the `django.views` module

View Class

- The **View class** serves as the **foundational master** class-based **base view** in Django
- All other class-based views **inherit** from this **base**
- It **defines** the **HTTP methods** that the view can handle
  - `['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']`

View Class

- As the **primary building block**, the **View** class provides
  - a structure for **handling** various types of **requests**
  - and forms the **basis** for the **implementation** of more specific class-based **views** in Django

```python
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):...
```

as_view() Method

- `as_view()` method is decorated by a `@classonlymethod`
- It is a **class-level** method and **cannot** be **accessed** through an instance of the class
- The method iterates over `initkwargs`, performing **validations** and **preparing** the view for **handling requests**

```
@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
```

view() Function

- Within the **as_view** method, the **encapsulated view function** accepts the standard parameters
  - `request`, `*args`, and `**kwargs`
- During its execution, it **binds** the **instance** of the **class** to the **class attributes** using **self**
- Specifically, it **associates self** with the **class attributes** provided in `initkwargs`
- Additionally, the function **assigns request** to `self.request`, **args** to `self.args`, and **kwargs** to `self.kwargs`

view() Function

- The **binding** ensures that the **view function** has **access** to the essential **request** and **argument** information
  - when handling **HTTP requests**
  - **facilitating** seamless **integration** with the class-based view

```
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 view function wraps around an instance of the class and executes the `dispatch()` method on that instance

dispatch() Method

- In Django's class-based views, the `dispatch()` method is a **fundamental** part of the **view processing flow**
- It is **responsible** for determining which **specific method** to **call** based on the **HTTP method** of the **incoming request**
- The `dispatch()` method **acts** as a kind of **router** for **directing** the **flow** of **control** to the appropriate **method**
  - E.g., **`get()`, `post()`**, etc. based on the type of **HTTP request**
  - You can **override** specific **methods** (e.g., **get**, **post**) to **customize** the **behavior** for different HTTP methods

Request-Response Lifecycle in CBVs

- The **lifecycle** of **request-response** in CBVs involves several stages, from the **initiation** of the **request** to the **generation** of the **response**

1. **Initialization**
2. **`as_view()` Method**
3. **`dispatch()` Method**
4. **Pre-processing (Optional)**
5. **HTTP Method-Specific Methods**
6. **View Logic Execution**
7. **Response Generation**
8. **Post-processing (Optional)**
9. **Response Sent to Client**

Simple Generic Views

TemplateView

- The `TemplateView` is designed to **render** a **specified** template, **enriching** the **context** with parameters captured from the URL
- It **inherits** methods and attributes from three main classes
  - **`TemplateResponseMixin`**
  - **`ContextMixin`**
  - **`View`**

```
class TemplateView(TemplateResponseMixin, ContextMixin, View):
    """
    Render a template. Pass keyword arguments from the URLconf to the context.
    """
```

Basic TemplateView Example

```python
class IndexViewWithTemplate(views.TemplateView):
    template_name = 'index.html'
    extra_context = {
        'title': 'Template view',
    }
```

```python
from django.urls import path

from cbv.web.views import IndexView, IndexViewWithTemplate


urlpatterns = (
    path('', IndexViewWithTemplate.as_view()),
)
```

RedirectView

- The **RedirectView** in Django is designed to perform **redirects** to a **specified** URL
- It **inherits** from the base **View** class and returns an **HttpResponsePermanentRedirect**

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/'))
]
```

Form Views

Generic Editing Views

FormViews

- **Form Views** are a **key** component of **handling form submissions**
- They are **responsible** for **rendering** forms to users
  - **Processing** the **data** **submitted** in the form
  - **Performing** **actions** based on that **data**
- They play a **crucial** role in **creating**, **updating**, and **validating** data **through forms**

Generic Editing Views

- In Django, `CreateView`, `UpdateView`, and `DeleteView` are class-based views that
  - provide **convenient** ways to **handle** the **Create**, **Update**, and **Delete** (**CUD**) operations
- These views are **part** of Django's **generic** class-based views and can be used to
  - **quickly** set up views for **common operations**

Generic Editing Views (CUD Operations)

- 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

```
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

success_url and reverse_lazy()

- `success_url` **attribute**
  - In a class-based view, such as `CreateView` or `UpdateView`, the `success_url` **attribute** is used to **specify** the URL to which the user should be **redirected** after a **successful** form submission
  - This attribute can be **set** to a URL **string** or a URL **pattern name**
- `reverse_lazy()`
  - It is a **utility function** provided by Django to **reverse** URL **patterns** at a **lazy**, or **deferred**, time
  - It is used to **obtain** the URL for a given view or URL **pattern name**
  - It's particularly useful in situations where the URL configuration is **not fully loaded** when the module is imported (such **as in class-based views**)

get_success_url() Method

- The `get_success_url` method is used to **determine dynamically** the **URL** to which the user should be **redirected** after a **successful** form **submission** or another **successful** operation

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

    def get_success_url(self):
        return reverse_lazy('details', kwargs={'pk':
self.object.pk})
```

Redirects to the detail view of the updated object

ModelFormMixin

- In Django, `ModelFormMixin` is a **mixin** class that provides functionality
  - to work with **model forms** in **class-based views**
- It is commonly used in **conjunction** with **generic class-based** views
  - `CreateView`, `UpdateView`, and `DeleteView`
- It **simplifies** the process of working with **forms** associated with **model** instances

ModelFormMixin Usage

- Using **ModelFormMixin** with **CreateView** to create a new object

```
from django.views.generic.edit import CreateView
from .models import Article
from .forms import ArticleForm

class ArticleCreateView(CreateView):
    model = Article
    form_class = ArticleForm   # Specifies the form class
associated with the model
    template_name = 'create_article.html'
    success_url = '/success/'  # Redirects to this URL upon
successful form submission
```

Customizing Form Handling

- There are **key methods** you can **override** to **customize** form handling in CBVs

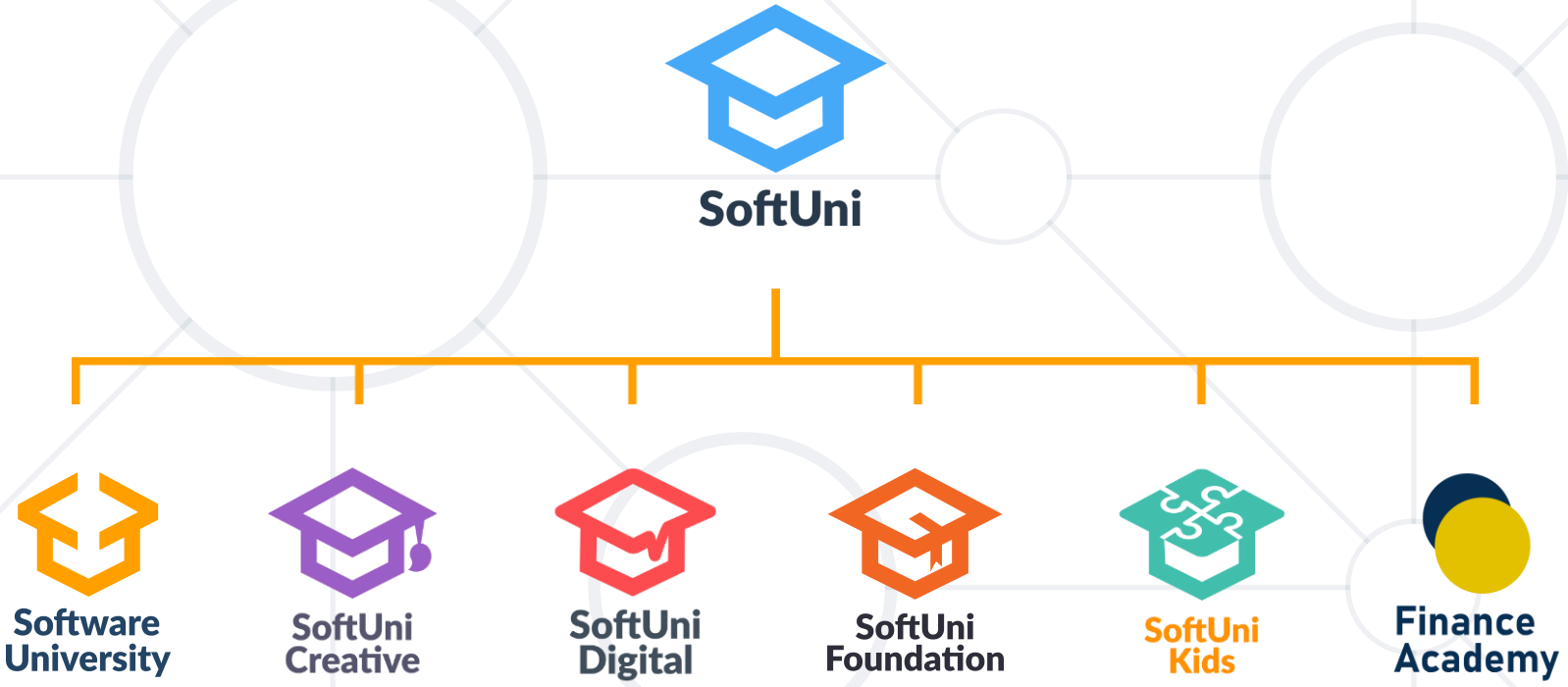
```
class ArticleCreateView(CreateView):
    ...
    def form_valid(self, form):
        # Custom logic for valid forms during object creation
        return super().form_valid(form)

    def form_invalid(self, form):
        # Custom logic for invalid forms during object creation
        return super().form_invalid(form)
```

Summary

- **Class-Based** Views
- **Base** Views
  - View Class
- Simple **Generic** Views
  - TemplateView, RedirectView
- **Form** Views
  - CreateView, UpdateView, DeleteView

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank
Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR
Serving the high achievers

AMBITIONED

PHAR VISION

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

- 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