Функции представлений

В Django представления можно создать на основе классов (CBV) или на основе функций (FBV).

Одним из основных применений Django является предоставление HTTP-ответов в ответ на HTTP-запросы. Django позволяет делать это с помощью так называемых представлений. Представление - это просто вызываемый объект, который принимает запрос и возвращает ответ.

Django изначально поддерживал только представления на основе функций (FBV), но их было трудно расширять, они не использовали преимущества объектно-ориентированного программирования (ООП) и не были DRY. Именно поэтому разработчики Django решили добавить поддержку представлений на основе классов (CBVs). CBV используют принципы ООП, что позволяет использовать наследование, повторно использовать код и в целом писать более качественный и чистый код.

Django предлагает готовые или общие CBV, которые обеспечивают решение общих проблем. Они имеют удобные для программистов названия и предлагают решения таких проблем, как отображение данных, редактирование данных и работа с данными на основе даты. Они могут использоваться самостоятельно или наследоваться в пользовательских представлениях.

Представления на основе функций (FBV)
По своей сути FBVs - это просто функции. Их легко читать и работать с ними, поскольку вы можете видеть, что именно происходит.

Плюсы
- Явный поток кода (у вас есть полный контроль над тем, что происходит)
- Проста в реализации
- Легко понять
- Отлично подходит для уникальной логики представления
- Легко интегрировать с декораторами

Минусы
- Много повторяющегося кода
- Обработка HTTP методов через условное ветвлен
- Не используют преимущества ООП
- Труднее поддерживать

```
urlpatterns = [
    path('create/', task_create_view, name='task-create'),
]
```

```
from django.shortcuts import render, redirect
from django.views import View


def task_create_view(request):
    if request.method == 'POST':
        form = TaskForm(data=request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('task-list'))

    return render(request, 'todo/task_create.html', {
        'form': TaskForm(),
    })
```

ModelForm

Если создается приложение, управляемое базой данных, то, скорее всего, будут формы, которые тесно связаны с моделями Django. Например, у вас может быть модель BlogComment, и вы хотите создать форму, позволяющую людям оставлять комментарии. В этом случае было бы излишним определять типы полей в форме, потому что вы уже определили поля в модели.

По этой причине Django предоставляет вспомогательный класс, который позволяет вам создать класс Form из модели Django.

Например:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

пример

```python
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = [
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
]

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
```

```python
class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']
```

```python
class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

```python
class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter', 'slug']
        field_classes = {
            'slug': MySlugFormField,
        }
```

Todo App (использование FBVs)

Создадим небольшой проект, определим модели, создадим HTML-шаблоны и views.py.

```python
# todo/views.py

from django.shortcuts import render, get_object_or_404, redirect

from .forms import TaskForm, ConfirmForm
from .models import Task


def task_list_view(request):
    return render(request, 'todo/task_list.html', {
        'tasks': Task.objects.all(),
    })


def task_create_view(request):
    if request.method == 'POST':
        form = TaskForm(data=request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('task-list'))

    return render(request, 'todo/task_create.html', {
        'form': TaskForm(),
    })


def task_detail_view(request, pk):
    task = get_object_or_404(Task, pk=pk)
    return render(request, 'todo/task_detail.html', {
        'task': task,
    })
```

```python
def task_update_view(request, pk):
    task = get_object_or_404(Task, pk=pk)

    if request.method == 'POST':
        form = TaskForm(instance=task, data=request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('task-detail', args={pk: pk}))

    return render(request, 'todo/task_update.html', {
        'task': task,
        'form': TaskForm(instance=task),
    })


def task_delete_view(request, pk):
    task = get_object_or_404(Task, pk=pk)

    if request.method == 'POST':
        form = ConfirmForm(data=request.POST)
        if form.is_valid():
            task.delete()
            return HttpResponseRedirect(reverse('task-list'))

    return render(request, 'todo/task_delete.html', {
        'task': task,
        'form': ConfirmForm(),
    })
```

Представления на основе классов (CBV)

Виды на основе классов, которые были введены в Django 1.3, предоставляют альтернативный способ реализации представлений в виде объектов Python вместо функций. Они позволяют использовать принципы ООП (самое главное - наследование). Можно использовать CBVs для обобщения частей нашего кода и извлечения их в виде представлений суперкласса.

Плюсы

- Являются расширяемыми
- Они используют преимущества концепций ООП (самое главное - наследование)
- Отлично подходят для написания CRUD представлений
- Более чистый и многократно используемый код
- Встроенные в Django общие CBVs
- Они похожи на представления REST фреймворка Django

Минусы

- Неявный поток кода (многое происходит в фоновом режиме)
- Используется много миксинов, что может запутать
- Более сложный и трудный для освоения
- Декораторы требуют дополнительного импорта или переопределения кода

Перепишем предыдущий пример FBV как CBV:

```python
from django.shortcuts import render, redirect
from django.views import View


class TaskCreateView(View):

    def get(self, request, *args, **kwargs):
        return render(request, 'todo/task_create.html', {
            'form': TaskForm(),
        })

    def post(self, request, *args, **kwargs):
        form = TaskForm(data=request.POST)
        if form.is_valid():
            task = form.save()
            return redirect('task-detail', pk=task.pk)

        return self.get(request)
```

этот пример не сильно отличается от подхода FBV. Логика более или менее одинакова. Основное отличие заключается в организации кода. Здесь каждый HTTP-метод рассматривается отдельным методом вместо условного ветвления. В CBV можно использовать следующие методы: get, post, put, patch, delete, head, options, trace.

Еще одним плюсом такого подхода является то, что HTTP-методы, которые не определены, автоматически возвращают 405 Method Not Allowed ответ.

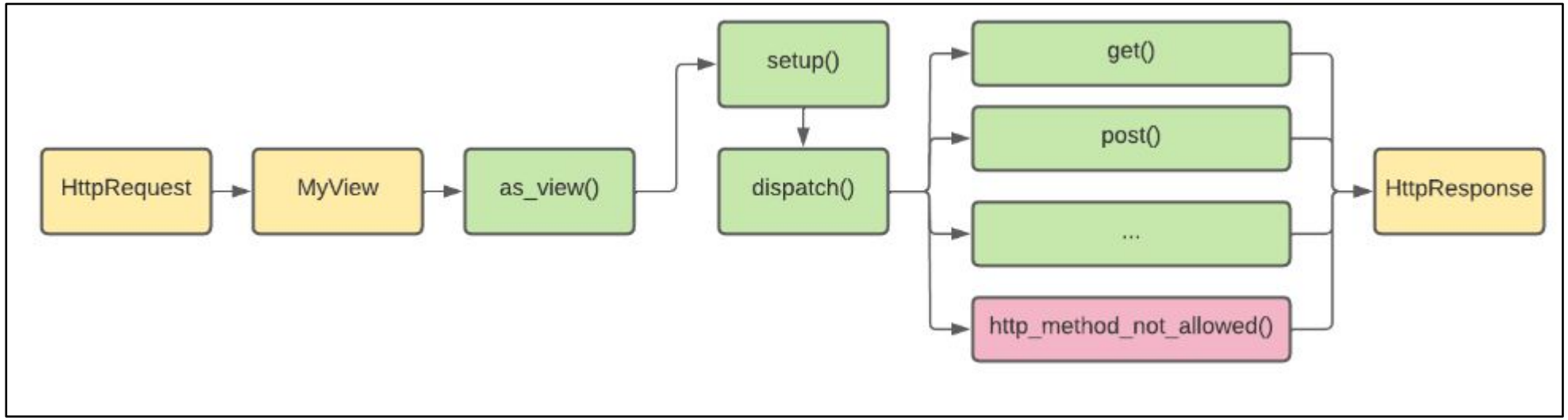
Поскольку URL-резольвер Django ожидает вызываемую функцию, то нужно вызвать as_view() при регистрации их в urls.py:

```python
urlpatterns = [
    path('create/', TaskCreateView.as_view(), name='task-create'),
]
```

Поток кода

Поток кода для CBV немного сложнее, потому что некоторые вещи происходят в фоновом режиме. Если мы расширим базовый класс View, то будут выполнены следующие шаги кода:

- Диспетчер URL Django направляет HttpRequest на MyView.
- Диспетчер URL Django вызывает as_view() на MyView.
- as_view() вызывает setup() и dispatch().
- dispatch() вызывает метод для определенного метода HTTP или http_method_not_allowed().
- Возвращается HttpResponse.



Перепишем наше приложение todo, чтобы использовать только CBVs:

```python
# todo/views.py

from django.shortcuts import render, get_object_or_404, redirect
from django.views import View

from .forms import TaskForm, ConfirmForm
from .models import Task


class TaskListView(View):

    def get(self, request, *args, **kwargs):
        return render(request, 'todo/task_list.html', {
            'tasks': Task.objects.all(),
        })


class TaskCreateView(View):

    def get(self, request, *args, **kwargs):
        return render(request, 'todo/task_create.html', {
            'form': TaskForm(),
        })

    def post(self, request, *args, **kwargs):
        form = TaskForm(data=request.POST)
        if form.is_valid():
            task = form.save()
            return redirect('task-detail', pk=task.pk)

        return self.get(request)


class TaskDetailView(View):

    def get(self, request, pk, *args, **kwargs):
        task = get_object_or_404(Task, pk=pk)

        return render(request, 'todo/task_detail.html', {
            'task': task,
        })
```

```python
class TaskUpdateView(View):

    def get(self, request, pk, *args, **kwargs):
        task = get_object_or_404(Task, pk=pk)
        return render(request, 'todo/task_update.html', {
            'task': task,
            'form': TaskForm(instance=task),
        })

    def post(self, request, pk, *args, **kwargs):
        task = get_object_or_404(Task, pk=pk)
        form = TaskForm(instance=task, data=request.POST)
        if form.is_valid():
            form.save()
            return redirect('task-detail', pk=task.pk)

        return self.get(request, pk)


class TaskDeleteView(View):

    def get(self, request, pk, *args, **kwargs):
        task = get_object_or_404(Task, pk=pk)
        return render(request, 'todo/task_confirm_delete.html', {
            'task': task,
            'form': ConfirmForm(),
        })

    def post(self, request, pk, *args, **kwargs):
        task = get_object_or_404(Task, pk=pk)
        form = ConfirmForm(data=request.POST)
        if form.is_valid():
            task.delete()
            return redirect('task-list')

        return self.get(request, pk)
```

сделаем urls.py вызывающим as_view():

```
# todo/urls.py

from django.urls import path

from .views import TaskListView, TaskDetailView, TaskCreateView, TaskUpdateView,
TaskDeleteView


urlpatterns = [
    path('', TaskListView.as_view(), name='task-list'),
    path('create/', TaskCreateView.as_view(), name='task-create'),
    path('<int:pk>/', TaskDetailView.as_view(), name='task-detail'),
    path('update/<int:pk>/', TaskUpdateView.as_view(), name='task-update'),
    path('delete/<int:pk>/', TaskDeleteView.as_view(), name='task-delete'),
]
```

Больше не используется условное ветвление. Если посмотреть на TaskCreateView и TaskUpdateView, то увидим, что они практически одинаковы. Можно еще больше улучшить этот код, извлекая общую логику в родительский класс. Кроме того, можно извлечь логику представления и использовать ее в представлениях для других моделей. Именно на таких изменениях основано видов на общих классах

Рассмотрим пример:

```python
from django.views.generic import CreateView


class TaskCreateView(CreateView):
    model = Task
    context_object_name = 'task'
    fields = ('name', 'description', 'is_done')
    template_name = 'todo/task_create.html'
```

Мы создали класс с именем TaskCreateView и унаследовали от него CreateView. Тем самым мы получили много функциональности, почти без кода. Теперь нам просто нужно установить следующие атрибуты:

model определяет, с какой моделью Django работает представление.
fields используется Django для создания формы (альтернативно, мы могли бы предоставить form_class).
template_name определяет, какой шаблон использовать (по умолчанию /<app_name>/<model_name>_form.html).
context_object_name определяет контекстный ключ, под которым экземпляр модели передается шаблону (по умолчанию object).
success_url определяет, куда пользователь будет перенаправлен при успехе (альтернативно, вы можете установить get_absolute_url в вашей модели).

Встроенные типы CBV в Django

Generic Display Views

Designed to display data.

- DetailView
- ListView

https://django.fun/ru/docs/django/4.0/ref/class-based-views/generic-display/#detailview

Generic Editing Views

Provide a foundation for editing content.

- FormView
- CreateView
- UpdateView
- DeleteView

https://django.fun/ru/docs/django/4.0/ref/class-based-views/generic-editing/#formview

Generic Date-based Views

Allow in-depth displaying of date-based data.

- ArchiveIndexView
- YearArchiveView
- MonthArchiveView
- WeekArchiveView
- DayArchiveView
- TodayArchiveView
- DateDetailView

https://django.fun/ru/docs/django/4.0/ref/class-based-views/generic-date-based/#archiveindexview

Задание

Изменить функции представлении с модели FBV на CBV в проекте с постами