장고 핵심 기능 - View

제네릭 뷰 선택

❖ 클래스형 뷰

- 클래스형 제너릭 뷰를 상속
- 필요한 속성, 메서드 오버라이딩

❖ 제너릭 뷰

- 공통적으로 사용하는 로직을 미리 개발해놓고 기본 클래스로 제공

제네릭 뷰 선택

❖ 제너릭 뷰 요약

제네릭 뷰 선택

❖ **View**

- 모든 클래스형 뷰의 기본이 되는 최상위 뷰
- 모든 클래스형 뷰는 이 클래스를 상속

```
class TestView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

제네릭 뷰 선택

❖ TemplateView

- 단순하게 화면에 보여줄 템플릿 파일을 처리하는 간단한 뷰
- 템플릿 파일만 지정하면 됨

```
class HomeView(TemplateView):
    template_name = 'home.html'
```

제네릭 뷰 선택

❖ DetailView

- 특정 객체 하나에 대한 정보를 보여주는 뷰
- 기본키로 지정된 레이코드 하나에 대한 정보를 보여줌
- 디폴트 컨텍스트 변수 : object

```
class PostDV(DetailView):
    model = Post
```

- url 패턴에 pk 항목(또는 slug)이 반드시 제시되야 함

```
re_path(r'^post(?P<slub>[-\w]+)/$', views.PostDV.as_view(),
        name='post_detail')
```

제네릭 뷰 선택

❖ ListView

- 여러 객체의 리스트를 보여주는 뷰
- 테이블의 모든 레코드에 대한 목록 보여주기
- object_list 컨텍스트 변수에 레코드들이 저장되어 템플릿에 전달

```
class PostLV(ListView):
    model = Post
```

제네릭 뷰 선택

❖ FormView

- 폼을 보여주는 제너릭 뷰
- 주요 속성
  - `form_class` : 폼 클래스 지정
  - `template_name` : 폼을 렌더링하는 템플릿
  - `success_url` : 폼 처리 성공 후 리다이렉트할 목적지

- 주요 메소드
  - `get()` : GET 요청 처리
  - `post()`: POST 요청 처리

제네릭 뷰 선택

❖ FormView

```
class SearchFormView(FormView):
    form_class = PostSearchForm
    template_name = 'blog/post_search.html'

    def form_valid(self, form):
        searchWord = form.cleaned_daa['search_word']
        post_list = Post.objects.filter(Q(title_icontains = searchWord) |
                                Q(description_icontains=searchWord) |
                                Q(content_icontains=searchWord).distinct()

        context = {}
        context['form'] = form
        context['search_term'] = searchWord
        context['object_list'] = post_list

        return render(self.request, self.template_name, context)
```

제네릭 뷰 선택

❖ CreateView

- 새로운 레코드를 생성해서 테이블에 저장해주는 뷰
- 새로운 레코드 생성을 위해 레코드 정보를 입력 받을 수 있는 폼이 필요
- 포함된 기능
  - FormView의 기능
  - 모델 정의로부터 폼을 자동으로 생성해주는 기능
  - 레코드를 정하는 기능

```
class PostCreateView(LoginRequiredMixin, CreateView):
    model = Post
    fields = ['title', 'slug', 'description', 'content', 'tags']
    initial = { 'slug' : 'auto-filling-do-not-input'}
    success_url = reverse_lazy('blog:index')

    def form_valid(self, form):
        form.instance.owner = self.request.user  # 현재 로그인 사용자로 설정
        return super().form_valid(form)
```

제네릭 뷰 선택

❖ UpdateView

- 테이블에 이미 있는 레코드를 수정하는 제너릭 뷰
- CreateView의 기능과 유사
- FormView의 기능 포함
- 작업 대상 테이블로부터 폼을 만들어 주며,
- 최종적으로 테이블에 있는 기존 레코드를 수정

```
class PostUpdateView(LoginRequiredMixin, UpdateView):
    model = Post
    field = ['title', 'slug', 'description', 'content', 'tag']
    success_url = reverse_lazy('blog:index')
```

- 작업 대상에 대한 pk 또는 slug가 url 패턴에 있어야 함

```
# blog/99/update
path('<int:pk>/update', views.PostUpdateView.as_view(), name='update)
```

제네릭 뷰 선택

❖ DeleteView

- 기존 객체를 삭제하기위한 제너릭 뷰
- success_url 속성에 삭제 후 이동할 url 지정

```
class PostDeleteView(LoginRequiredMixin, DeleteView):
    model = Post
    success_url = reverse_lazy('blog:index')
```

- 삭제 대상이 되는 pk 또는 slug 정보가 url 패턴이 있어야 함

```
# Example: /blog/99/delete
path('<int:pk>/delete/', views.PostDeleteView.as_view(), name='delete')
```

제네릭 뷰 선택

❖ ArchiveIndexView

- 날짜를 기준으로 리스팅해주는 뷰
- 날짜 기반 제너릭 뷰의 최상위 뷰
- 대상이 되는 모든 객체를 날짜 기준 내림 차순으로 정렬
- date_field 속성으로 기준 필드 설정

```
class PostAV(ArchiveIndexView):
    model = Post
    date_field = 'modify_dt'
```

- 컨텍스트 변수
  - object_list

제네릭 뷰 선택

❖ YearArchiveView

- 연도가 주어지면 이에 속하는 객체를 대상으로 가능한 월을 알려주는 제너릭 뷰
- 디폴트 동작
  - 객체들을 출력하는 것이 아님
  - 객체의 날짜 필드를 조사하여 월을 추출
  - 주어진 연도에 해당하는 개체들을 알고 싶다면 make_object_list 속성을 True로 지정
- model과 date_field 속성 지정

```
class PostYAV(YearArchiveView):
    model = Post
    date_field = 'modify_dt'
    make_object_list = True
```

- 대상 연도는 URLconf에서 year 변수에서 추출

```
# Example: /blog/archive/2019/
path('archive/<int:year>/', views.PostYAV.as_view(),
name='post_year_archive'), ... )
```

제네릭 뷰 선택

❖ MonthArchiveView

- 주어진 연/월에 해당하는 객체를 보여주는 제너릭 뷰
- 연/월 인자는 URLconf에서 지정
- model, date_field 속성 지정.
- make_object_list 속성은 없음

```
class PostMAV(MonthArchiveView):
    model = Post
    date_field = 'modify_dt'
```

- URLconf

```
# Example : /blog/archive2019/nov
patch('archive/<int:year>/<str:month>/', views.PostMAV.as_view(), ... )
```

제네릭 뷰 선택

❖ WeekArchiveView

- 연도와 주가 주어지면 그에 해당하는 객체를 보여주는 객체
- 연/주 인자는 URLconf에서 지정
- model, date_field 속성 지정

```
class TestWeekArchiveView(WeekArchiveView):
    model = Post
    date_field = 'modify_qt'
```

- URLconf

```
# Example: /blog/archive/2019/week/23
path('archive/<int:year>/week/<int:week>/',
     Views.TestWeekArchiveView.as_view(), ...)
```

제네릭 뷰 선택

❖ DayArchiveView

- 연/월/일이 주어지면 그에 해당하는 객체를 보여주는 제너릭 뷰
- 연/월/일 인자는 URLconf에서 지정
- model, date_field 속성 지정

```
class PostDAV(DayArciveView):
    model = Post
    date_field = 'modify_dt'
```

- URLconf

```
# Example: /blog/archive/2019/week/23
path('archive/<int:year>/week/<int:week>/',
      Views.TestWeekArchiveView.as_view(), ...)
```

제네릭 뷰 선택

❖ **TodayArchiveView**

- 오늘 날짜에 해당하는 객체를 보여주는 제너릭 뷰
- 오늘 날짜를 사용하므로 연/월/일 인자 필요 없음
- DayArchiveVeiw와 동일

```
class PostTAV(TodayArchiveView)
    model = Post
    date_field = 'modify_dt'
```

제네릭 뷰 선택

❖ DateDetailView

- 날짜 기준으로 특정 객체를 찾아서 그 객체의 상세 정보를 보여주는 뷰
- DetailView와 유사하지만 객체를 찾는데 사용하는 안지로 연/월/일 정보를 추가적으로 사용

→ pk또는 slug도 같이 사용

```
class TestDateDetailView(DateDetailView):
    model = Post
    date_field = 'modify_qt'
```

- URLconf

```
# Example: /blog/archive/2019/nov/10/99/
path('archive/<int:year>/<str:month>/<int:day>/<int:pk>/',
       views.TestDateDetaiView.as_view(), ...)
```

제네릭 뷰 오버라이딩

❖ 속성 오버라이딩

- **model**
  - 기본 뷰(View, TemplateView, RedirectView)와 FormView를 제외한 모든 제너릭뷰에서 사용
  - 작업 대상 데이터가 들어 있는 모델을 지정
  - model 대신 queryset 속성 지정 가능

```
model = Bookmark
queryset = Bookmark.objects.all()
```

- **queryset**
  - 기본 뷰(View, TemplateView, RedirectView)와 FormView를 제외한 모든 제너릭뷰에서 사용
  - 작업 대상이 되는 QuerySet 객체를 지어
  - queryset 속성을 지정하면 model 속성은 무시됨

- **template_name**
  - TemplateView를 포함한 모든 제너릭뷰에서 사용하는 속성
  - 템플릿 파일명을 문자열로 지정

제네릭 뷰 오버라이딩

❖ 속성 오버라이딩

- **context_object_name**
  - 기본 뷰(View, TemplateView, RedirectView)를 제외한 모든 제너릭 뷰에서 사용
  - 템플릿 파일에서 사용할 컨텍스트 변수명을 지정

- **paginate_by**
  - ListView와 날짜 기반 뷰에서 사용
  - 페이지당 몇 개 항목을 출력할지 정수로 지정
  - 페이징 기능을 활성화

- **date_field**
  - 날짜 기반 뷰에서 기준이 되는 필드를 지정
  - 이 필드를 기준으로 년/월/일 검사

- **make_object_list**
  - YearArchiveView 사용 시 해당 년에 맞는 객체들의 리스트를 생성할지 여부 지정
  - True이면 객체들의 리스트를 만들고 템플릿에서 사용가능
  - False이면 qeuryset 속성에 None이 할당

제네릭 뷰 오버라이딩

❖ 속성 오버라이딩

- **form_class**
  - FormView, CreateView, UpdateView에서 사용
  - 폼을 만드는데 사용할 클래스 지정

- **initial**
  - FormView, CreateView, UpdateView에서 사용
  - 폼에 사용할 초기 데이터를 사전으로 지정

- **fields**
  - CreateView, UpdateView에서 사용
  - 폼에 사용할 필드를 지정
  - ModelForm 클래스의 Meta.fields 속성과 동일한 의미

- **success_url**
  - FormView, CreateView, UpdateView, DeleteVeiw에서 사용
  - 폼에 대한 처리가 성공한 후 리다이렉트할 URL 지정

제네릭 뷰 오버라이딩

❖ 메서드 오버라이딩

- **get_queryset()**
  - 기본 뷰(View, TemplateView, RedirectView)와 FormView를 제외한 모든 제너릭에서 사용
  - 출력 객체를 검색하기 위한 대상 QuerySet 객체 또는 출력 대상인 객체 리스트를 반환
  - 디폴트는 queryset 속성 값을 반환
  - queryset 속성이 지정되지 않은 경우
    - 모델 매니저 클래스의 all() 메서드를 호출해서 QuerySet 객체를 생성하고 이를 반환

- **get_context_data(**kwargs)**
  - TemplateView를 포함해서 모든 제너릭 뷰에서 사용하는 메서드
  - 템플릿에서 사용할 컨텍스틀 데이터를 반환

- **form_valid(form)**
  - FormView, CreateView, updateView에서 사용
  - get_success_url() 메서드가 반환하는 URL로 리다이렉트 수행

본문의 예제

```
class TestPostLV(ListView)
    model = Post
    template_name = 'blog/post_all.html'
    context_object_name = 'posts'
    paginate_by = 2
```

본문의 예제

```
class TestPostLV(ListView)
    # model = Post
    queryset = Post.objects.all()[:5]  # queryset 지정시 model 속성 불필요
    template_name = 'blog/post_all.html'
    context_object_name = 'posts'
    paginate_by = 2
```

본문의 예제

❖ template_name 속성

```
class TestPostLV(ListView)
    # model = Post
    # queryset = Post.objects.all()[:5]   # queryset 지정시 model 속성 불필요
    template_name = 'blog/post_all.html'
    context_object_name = 'posts'
    paginate_by = 2

    def get_queryset(self):
        return Post.objects.filter(
                Q(content_icontains=self.kwargs['word'])).distinct()
```

본문의 예제

```
class TestPostLV(ListView)
    # model = Post
    # queryset = Post.objects.all()[:5]   # queryset 지정시 model 속성 불필요
    # template_name = 'blog/post_all.html'
    template_name = 'blog/post_test.html'

    context_object_name = 'posts'
    paginate_by = 2

    def get_queryset(self):
        return Post.objects.filter(
           Q(content_icontains=self.kwargs['word'])).distinct()
```

본문의 예제

```
class TestPostLV(ListView)
    # model = Post
    # queryset = Post.objects.all()[:5]   # queryset 지정시 model 속성 불필요
    # template_name = 'blog/post_all.html'
    template_name = 'blog/post_test.html'

    context_object_name = 'posts'
    paginate_by = 2

    def get_queryset(self):
        return Post.objects.filter(
              Q(content_icontains=self.kwargs['word'])).distinct()

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

제네릭 뷰의 처리 흐름

❖ **ListView**

- 클라이언트의 HTTP 요청 GET 메서드에 따라 get() 메인 메서드가 실행
- 대상 테이블로부터 조건에 맞는 복수 개의 객체를 가져옴
- 필요하다면 이 객체들에 추가 로직을 반영
- 최종 결과 객체 리스트를 object_list라는 컨텍스트 변수에 넣어 템플릿에 전달
- 템플릿 파일에 따라, 최종 HTML 응답을 만들고 이를 클라이언트에 응답

제네릭 뷰의 처리 흐름

메서드	설명
setup()	공통으로 사용할 속성들을 미리 정의하는 메서드 self.request, self.args, self.kwargs는 미리 정의됨 그 외에 추가할 것이 있으면 오버라이딩
dispatch()	클라이언트 요청의 HTTP 메소드를 검사해, 뷰클래스에 정의된 적절한 처리 메서드를 호출 GET/POST 요청에 따라 get()/post() 메서드 호출
get()	메인 처리 메서드
http_method_not_allowed()	get, post 등의 메인 처리 메서드를 찾지 못한 경우 호출됨
get_queryset()	작업 대상 객체들의 리스트를 반환 리스트는 QuerySet 객체와 같은 순환가능한 객체
get_context_data()	템플릿에서 사용할 컨텍스트 데이터를 반환하는 메서드 get_context_object_name() 메서드를 호출
get_context_object_name()	템플릿에서 사용할 컨텍스트 명(context_object_name 속성) 반환 context_object_name 속성 생략시 모델명소문자_list가 됨 모델명이 Bookmark인 경우 bookmark_list
render_to_response()	최종 응답인 self.response_class 객체를 반환 get_template_names() 메서드를 호출
get_template_names()	템플릿 파일명을 담은 리스트를 반환

제네릭 뷰의 처리 흐름

❖ DetailView

- 클라이언트의 HTTP 요청 GET 메서드에 따라 get() 메인 메서드가 실행
- 대상 테이블로부터 조건에 맞는 객체 하나를 가져옴
- 필요하다면 이 객체에 추가 로직을 반영
- 최종 결과 객체를 object라는 컨텍스트 변수에 넣어 템플릿에 전달
- 템플릿 파일에 따라 최종 HTML 응답을 만들고 이를 클라이언트에 응답

제네릭 뷰의 처리 흐름

메서드	설명
setup()	DetailView와 동일
dispatch()	DetailView와 동일
get()	DetailView와 동일
http_method_not_allowed()	DetailView와 동일
get_queryset()	작업 대상 객체들의 리스트를 반환 리스트는 QuerySet 객체와 같은 순환가능한 객체
get_object()	**작업 대상 객체 하나를 반환** **먼저 get_queryset()을 호출해 검색 대상 객체 리스트를 얻음** **검색 시 pk(없는 경우 slug)로 검색**
get_context_data()	템플릿에서 사용할 컨텍스트 데이터를 반환하는 메서드 get_context_object_name() 메서드를 호출
get_context_object_name()	템플릿에서 사용할 컨텍스트 명(context_object_name 속성) 반환 context_object_name 속성 생략시 모델명소문자_list가 됨 모델명이 Bookmark인 경우 bookmark_list
render_to_response()	최종 응답인 self.response_class 객체를 반환 get_template_names() 메서드를 호출
get_template_names()	템플릿 파일명을 담은 리스트를 반환

제네릭 뷰의 페이징 처리

❖ 페이징 기능 활성화

- paginate_by 속성
  - 페이지당 객체의 개수

- 페이지 정보를 url 패턴에서 추출

```
# /objects/page3
path('objects/page<int:page>/', PaginatedView.as_view()), ..)
```

- 페이지 정보는 get 파라미터에서 추출

```
/objects/?page=3

request.GET.get('page')
```

- 페이징 관련 컨텍스트 변수
  - object_list : 화면에 보여줄 객체의 리스트
  - is_paginated: 출력 결과가 페이징 처리가되는지 여부
  - paginator: django.core.paginator.Paginator
  - page_obj: django.core.paginator.Page 클래스

제네릭 뷰의 페이징 처리

❖ Paginator 클래스

- 페이징 기능의 메인 클래스
- 객체의 리스트와 페이지당 항목수를 필수 인자로 받아 각 페이지 객체를 생성

- `class Paginator(object_list, per_page, orphans=0, allow_empty_first_page=True)`
  - `object_list` : 페이징 대상 객체 리스트
  - `per_page` : 페이지당 객체 수
  - `orphans` : 마지막 페이지의 항목수가 이 값보다 작으면 이전 페이지 끝에 보여줌
  - `allow_empty_first_page`: 첫 페이지가 비어 있어도 되는지 여부

- 속성
  - `Paginator.count` : 항목의 총 개수
  - `Paginator.num_pages` : 페이지의 총 개수
  - `Paginator.page_range` : 1부터 시작하는 페이지 범위(예: `[1, 2, 3, 4]`)

- 주요 메서드
  - `Paginator.page(number)`
  - `Paginator.get_page(number)`
    - `number` 페이지(1부터 시작)에 해당하는 `Page` 객체 반환
    - `number`가 음수이거나 최대 페이지보다 크면 마지막 페이지 반환

제네릭 뷰의 페이징 처리

❖ Page 클래스

- Paginator 객체에 의해 생성된 단위 페이지를 나타내는 객체
- Paginator.page() 메서드 호출로 생성

- class Page(object_list, number, paginator)
  - object_list : Paginator 클래스의 object_list인자와 동일
  - number : 몇 번째 페이지인지를 지정하는 페이지 인덱스
  - paginator : 페이지를 생성해주는 Paginator 객체

- 속성
  - Page.object_list : 현재 페이지의 객체 리스트
  - Page.number : 현재 페이지의 번호
  - Page.paginator : 현재 페이지를 생성한 Paginator 객체

- 메서드
  - Page.has_next() : 다음 페이지가 있으면 True
  - Page.has_next() : 이전 페이지가 있으면 True
  - Page.has_other_pages() : 다음 또는 이전 페이지가 있으면 True
  - Page.next_page_number() : 다음 페이지 번호 반환, 없으면 InvalidPage 예외 발생
  - Page.previous_page_number() : 이전 페이지 번호 반환, 없으면 InvalidPage 예외 발생
  - Page.start_index() : 해당 페이지의 첫 번째 항목의 인덱스 반환
  - Page.end_index() : 해당 페이지의 마지막 항목의 인덱스 반환

제네릭 뷰의 페이징 처리

```
$ python manage.py shell

>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)
>>> p.count
4
>>> p.num_pages
2
>>> p.page_range
>>> range(1, 3)
>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']
>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
```

제네릭 뷰의 페이징 처리

```
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.previous_page_num()
1
>>> page2.start_index()
3
>>> page2.end_index()
4
```

단축 함수

❖ render_to_response() 단축 함수

- redner_to_response(template_name, context=None, content_type=None, status=None, using=None)
  - HttpResponse 객체를 반환
  - render() 함수로 대체됨

단축 함수

❖ render() 단축 함수

- redner(request, template_name, context=None, content_type=None, status=None, using=None)
  - 렌더링 처리 후 HttpResponse 객체를 반환

  - request
  - template_name : 템플릿 파일명
  - context : 템플릿 컨텍스트 데이터가 담긴 파이썬 사전형 객체
  - content_type : 최종 결과의 MIME 타입, 디플트는 DEFAULT_CONTENT_TYEP 설정 항목값
  - status : 응답 상태 코드, 디폴트 200
  - using: 사용할 템플릿 엔진 이름

단축 함수

❖ render() 단축 함수

```
from django.shortcuts import render

def my_view(request):
    # ...
    return render(request, 'myapp/index.html', {'foo': 'bar'},
content_type="application/xhtml+xml")
```

```
from django.shortcuts import render
from django.template import loader

def my_view(request):
    # ...
    t = loader.get_template('myapp/index.html')
    c = {'foo': 'bar'}
    return render(request, t, c, content_type="application/xhtml+xml")
```

단축 함수

❖ redirect() 단축 함수

- redirect(to, *args, permanent=False, **kwargs)
  - to 인자로 주어진 URL로 이동하기 위한 HttpResponseRedirect 객체를 반환
  - permanent인자가 True이면 영구 리다이렉션(301)을 하고, False(디폴트)면 임시 리다렉션(302, 307) 응답

  - to인자
    - 이동할 URL을 직접 지정

      ```
      return redirect(self.object.get_absolute_url())
      ```

    - 모델 지정 - 그 모델의 get_absolute_url() 메서드에서 반환하는 URL 지정

      ```
      return redirect(self.object)
      ```

    - URL 패턴의 이름 지정 - reverse() 함수를 호출하면서 그 패턴명을 넘김, 그 리턴값을 URL로 사용

      ```
      return redirect('photo:album_detail', pk=self.object.id)
      ```

단축 함수

❖ redirect() 단축 함수

```
class AlbumPhtoCV(LoginRequiredMixin, CreateView):
    :
    def form_valid(self, form):
        form.instance.owner = self.request.user
        context = self.get_context_data()
        formset = context['formset']

        for phtoform in formset:
            photoform.instance.owner = self.request.

        if formset.is_valid():
            self.object = form.save()
            formset.instance = self.object
            fromset.save()
            return redirect(self.get_success_url())
        else:
            return self.render_to_response(self.get_context_data(form=form))
```

단축 함수

❖ get_obejct_or_404() 단축 함수

- get_object_or_404(klass, *args, **kwargs)
  - klass 모델에 해당하는 테이블에서 args 또는 kwargs 조건에 맞는 레코드를 거백
  - 있으면 해당 레코드를 반환
  - 없으면 Http404 예외 발생
  - 조건에 맞는 레코드가 둘 이상이면 MultipleObjectsReturned 예외 발생
  - klass는 Model 또는 Manager 클래스, QuerySet 객체
  - 모델 매니저 클래스의 get() 메서드를 사용해 검색

단축 함수

❖ get_obejct_or_404() 단축 함수

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)


# 아래와 동일

from django.http import Http404

def my_view(request):
    try :
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No Model matches the given query.")
```

❖ get_lsit_or_404() 단축 함수

- get_list_or_404(klass, *args, **kwargs)
  - klass 모델에 해당하는 테이블에서 args 또는 kwargs 조건에 맞는 레코드를 검색
  - 있으면 해당 레코드의 리스트를 반환하고, 격과가 빈 리스트이면 Http404 예외 발생
  - klass는 Model 또는 Manager 클래스, QuerySet 객체

단축 함수

❖ get_lsit_or_404() 단축 함수

```
from django.shortcuts import get_list_or_404

def my_view(request)
   my_objects = get_list_or_404(MyModel, published=True)


# 아래와 동일

from django.http import Http404

def my_view(request)
    my_objects = list(MyModel.objects.filter(published=True)
    if not my_objects:
        raise Http404("No MyModel matches the given query")
```