첫 번째 장고 앱 작성하기, part 1

예제로 배워봅시다.

이 튜토리얼을 통해, 간단한 설문조사(Polls) 어플리케이션을 만드는 과정을 따라해 보겠습니다.

두 파트로 구성되어 있습니다.

- 사람들이 설문 내용을 보고 직접 투표할 수 있는 개방된 사이트
- 관리자가 설문을 추가, 변경, 삭제할 수 있는 관리용 사이트

Django 설치가 되어 있다고 가정합니다. 쉘 프롬프트 ($ 접두사로 표시)에서 다음 명령을 실행하여 Django가 설치되어 있고 어떤 버전인지 알 수 있습니다.

```
$ python -m django --version
```

Django가 설치 되었다면, 설치된 Django의 버전을 확인할 수 있습니다. 만약 설치가 제대로 되지 않았다면, 《No module named django》와 같은 에러가 발생합니다.

This tutorial is written for Django 3.1, which supports Python 3.6 and later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you're using an older version of Python, check 장고와 어떤 파이썬 버전을 사용해야 하나요? to find a compatible version of Django.

예전 버전의 Django를 제거하고 새로운 버전의 Django를 설치하는 방법은 Django 설치하기를 참조하세요.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

프로젝트 만들기

Django를 처음 사용한다면, 초기 설정에 주의를 기울여야 합니다. Django project를 구성하는 코드를 자동 생성해야 하는데, 이 과정에서 데이터베이스 설정, Django 위한 옵션들, 어플리케이션을 위한 설정들과 같은 Django 인스턴스를 구성하는 수많은 설정들이 생성되기 때문입니다.

커맨드라인에서 **cd** 명령으로 코드를 저장할 디렉토리로 이동 한 후, 다음의 명령을 수행합니다.

```
$ django-admin startproject mysite
```

이 명령은 현재 디렉토리에서 **mysite**라는 디렉토리를 생성할 것입니다. 만약 이 명령이 동작하지 않는다면, django-admin을 실행

하는 데 문제가 있습니다.를 확인하세요.

> **주석**
>
> 프로젝트를 생성할 때, Python 또는 Django에서 사용 중인 이름은 피해야 합니다. 특히, **`django`**(Django 그 자체와 충돌이 일어납니다)나, **`test`**(Python 패키지의 이름중 하나입니다) 같은 이름은 피해야 한다는 의미입니다.

> **코드가 어디에서 서비스 되어야 할까요?**
>
> 과거의 PHP 를 작성해본 경험이 있다면(최근의 프레임워크 말고) 아마 코드 전체를 **`/var/www`** 같은 웹 서버의 DocumentRoot 에 넣으려고 하려고 할겁니다. Django 에서는 그러지 마십시요. 파이선 코드가 웹서버의 DocumentRoot 에 존재하는것은 좋은 생각이 아닙니다. 웹을 통해서 외부의 사람들이 Python 코드를 직접 열어볼 수 있는 위험이 있기 때문입니다. 그렇게 되면 보안에 별로 좋지 않습니다.
>
> 작성한 코드를 **`/home/mycode`** 와 같은 DocumentRoot 의 **바깥에** 두는것을 권합니다.

`startproject` 에서 무엇이 생성되는지 확인해 봅시다.

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

이 파일들은,

- The outer **`mysite/`** root directory is a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- **`manage.py`**: Django 프로젝트와 다양한 방법으로 상호작용 하는 커맨드라인의 유틸리티 입니다. **`manage.py`** 에 대한 자세한 정보는 django-admin and manage.py 에서 확인할 수 있습니다.
- **`mysite/`** 디렉토리 내부에는 프로젝트를 위한 실제 Python 패키지들이 저장됩니다. 이 디렉토리 내의 이름을 이용하여, (**`mysite.urls`** 와 같은 식으로) 프로젝트의 어디서나 Python 패키지들을 임포트할 수 있습니다.
- **`mysite/__init__.py`**: Python으로 하여금 이 디렉토리를 패키지처럼 다루라고 알려주는 용도의 단순한 빈 파일입니다. Python 초심자라면, Python 공식 홈페이지의 패키지를 읽어보세요.
- **`mysite/settings.py`**: 현재 Django 프로젝트의 환경 및 구성을 저장합니다. Django settings에서 환경 설정이 어떻게 동작하는지 확인할 수 있습니다.
- **`mysite/urls.py`**: 현재 Django project 의 URL 선언을 저장합니다. Django 로 작성된 사이트의 《목차》 라고 할 수 있습니다. URL dispatcher 에서 URL 에 대한 자세한 내용을 읽어보세요.
- **`mysite/asgi.py`**: An entry-point for ASGI-compatible web servers to serve your project. See ASGI를 사용하여 배포하는 방법 for more details.
- **`mysite/wsgi.py`**: 현재 프로젝트를 서비스하기 위한 WSGI 호환 웹 서버의 진입점입니다. WSGI를 사용하여 배포하는 방법를 읽어보세요.

개발 서버

당신의 Django 프로젝트가 제대로 동작하는지 확인해 봅시다. **`mysite`** 디렉토리로 이동하고, 다음 명령어를 입력하세요.

```
$ python manage.py runserver
```

커맨드라인에서 다음과 같은 출력을 볼 수 있습니다.

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

3월 22, 2021 - 15:50:53
Django version 3.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

주석

현재 데이터베이스에 적용되지 않은 변경사항들(migrations)에 대한 경고들은 무시해도 됩니다. 데이터베이스에 대한 부분은 간단히 다루도록 하겠습니다.

Django 개발 서버를 시작했습니다. 개발 서버는 순수 Python으로 작성된 경량 웹 서버입니다. Django에 포함되어 있어 아무 설정 없이 바로 개발에 사용할 수 있습니다.

이쯤에서 하나 기억할 것이 있습니다. **절대로** 개발 서버를 운영 환경에서 사용하지 마십시오. **개발 서버는 오직 개발 목적으로만** 사용하여야 합니다(우리는 웹 프레임워크를 만들지 웹 서버를 만들지는 않거든요).

이제 서버가 동작하기 시작했으니, 자신의 웹 브라우져에서 http://127.0.0.1:8000/ 을 통해 접속할 수 있습니다. 로켓이 이륙하는 모습이 담긴 《Congratulations!》 페이지를 보게될 거에요. 잘 동작 하네요!

포트 변경하기

기본적으로, **runserver** 명령은 내부 IP 의 8000 번 포트로 개발 서버를 띄웁니다.

만약 이 서버의 포트를 변경하고 싶다면, 커맨드라인에서 인수를 전달해주면 됩니다. 예를들어, 이 명령은 포트를 8080 으로 서버를 시작할 것입니다.

```
$ python manage.py runserver 8080
```

서버의 IP를 변경하려면 포트와 함께 전달하십시오. 예를 들어, 사용 가능한 모든 공용 IP를 청취하려면 (이는 Vagrant를 실행 중이거나 네트워크의 다른 컴퓨터에서 작업하고 싶을 때 유용합니다) 다음을 사용하십시오.

```
$ python manage.py runserver 0:8000
```

** 0 ** **은 ** 0.0.0.0 **의 지름길입니다. 개발 서버의 전체 문서는 : djadmin :`runserver 〈참조에서 찾을 수 있습니다.

runserver 의 자동 변경 기능

개발 서버는 요청이 들어올 때마다 자동으로 Python 코드를 다시 불러옵니다. 코드의 변경사항을 반영하기 위해서 굳이 서버를 재기동 하지 않아도 됩니다. 그러나, 파일을 추가하는 등의 몇몇의 동작은 개발서버가 자동으로 인식하

지 못하기 때문에, 이런 상황에서는 서버를 재기동 해야 적용됩니다.

설문조사 앱 만들기

이제, 작업을 시작하기 위해 당신의 환경(프로젝트)이 설치되었습니다.

Django에서 당신이 작성하는 각 어플리케이션들은 다음과 같은 관례로 Python 패키지가 구성됩니다. Django 는 앱(app) 의 기본 디렉토리 구조를 자동으로 생성할 수 있는 도구를 제공하기 때문에, 코드에만 더욱 집중할 수 있습니다.

프로젝트 대 앱

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a small poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your Python path. In this tutorial, we'll create our poll app in the same directory as your **`manage.py`** file so that it can be imported as its own top-level module, rather than a submodule of **`mysite`**.

앱을 생성하기 위해 **`manage.py`**가 존재하는 디렉토리에서 다음의 명령을 입력해 봅시다.

```
$ python manage.py startapp polls
```

`polls`라는 디렉토리가 생겼습니다. 이걸 펼쳐놓으면 아래와 같습니다.

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

이 디렉토리 구조는 투표 어플리케이션의 집이 되어줄 것입니다.

첫 번째 뷰 작성하기

첫 번째 뷰를 작성해봅시다. 《polls/view.py》를 열어 다음과 같은 파이썬 코드를 입력합니다

polls/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

Django에서 가장 간단한 형태의 뷰입니다. 뷰를 호출하려면 이와 연결된 URL 이 있어야 하는데, 이를 위해 URLconf가 사용됩니다.

polls 디렉토리에서 URLconf를 생성하려면, **urls.py**라는 파일을 생성해야 합니다. 정확히 생성했다면, 앱 디렉토리는 다음과 같이 보일 겁니다.

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

《polls/urls.py》 파일에는 다음과 같은 코드가 포함되어 있습니다.

polls/urls.py

```
from django.urls import path

from . import views

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

다음 단계는, 최상위 URLconf 에서 **polls.urls** 모듈을 바라보게 설정합니다. **mysite/urls.py** 파일을 열고, **django.urls.include**를 import 하고, **urlpatterns** 리스트에 **include()** 함수를 다음과 같이 추가합니다.

mysite/urls.py

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

include() 함수는 다른 URLconf들을 참조할 수 있도록 도와줍니다. Django가 함수 **include()**를 만나게 되면, URL의 그 시점까지 일치하는 부분을 잘라내고, 남은 문자열 부분을 후속 처리를 위해 include 된 URLconf로 전달합니다.

include()에 숨은 아이디어 덕분에 URL을 쉽게 연결할 수 있습니다. polls 앱에 그 자체의 URLconf(**polls/urls.py**)가 존재하는 한, 《/polls/》, 또는 《/fun_polls/》, 《/content/polls/》와 같은 경로, 또는 그 어떤 다른 root 경로에 연결하더라도, 앱은 여전히 잘 동작할 것입니다.

언제 include()를 사용해야 하나요?

다른 URL 패턴을 포함할 때마다 항상 **include()**를 사용해야 합니다. **admin.site.urls**가 유일한 예외입니다.

이제 **index** 뷰가 URLconf에 연결되었습니다. 잘 작동하는지 확인하기 위해 다음 명령을 입력해 보세요.

```
$ python manage.py runserver
```

브라우저에서 http://localhost:8000/polls/를 입력하면 **index** 뷰에 정의한 《*Hello, world. You're at the polls index.*》 가 보일 것입니다.

페이지가 보이지 않나요?

에러 페이지가 표시된다면, http://localhost:8000/ 이 아니라 http://localhost:8000/polls/가 정확히 주소 창에 입력되었는지 확인하세요.

path() 함수에는 2개의 필수 인수인 **route** 와 **view**, 2개의 선택 가능한 인수로 **kwargs** 와 **name** 까지 모두 4개의 인수가 전달 되었습니다. 이 시점에서, 이 인수들이 무엇인지 살펴보는 것은 의미가 있습니다.

path() 인수: route

route 는 URL 패턴을 가진 문자열 입니다. 요청이 처리될 때, Django 는 **urlpatterns** 의 첫 번째 패턴부터 시작하여, 일치하는 패턴을 찾을 때 까지 요청된 URL 을 각 패턴과 리스트의 순서대로 비교합니다.

패턴들은 GET 이나 POST 의 매개 변수들, 혹은 도메인 이름을 검색하지 않습니다. 예를 들어,
https://www.example.com/myapp/ 이 요청된 경우, URLconf 는 오직 **myapp/** 부분만 바라 봅니다.
https://www.example.com/myapp/?page=3, 같은 요청에도, URLconf 는 역시 **myapp/** 부분만 신경씁니다.

---

path() 인수: view

Django 에서 일치하는 패턴을 찾으면, **HttpRequest** 객체를 첫번째 인수로 하고, 경로로 부터 〈캡처된〉 값을 키워드 인수로하여 특정한 view 함수를 호출합니다. 나중에 이에 대한 간단한 예제를 살펴보겠습니다.

---

path() 인수: kwargs

임의의 키워드 인수들은 목표한 view 에 사전형으로 전달됩니다. 그러나 이 튜토리얼에서는 사용하지 않을겁니다.

---

path() 인수: name

URL 에 이름을 지으면, 템플릿을 포함한 Django 어디에서나 명확하게 참조할 수 있습니다. 이 강력한 기능을 이용하여, 단 하나의 파일만 수정해도 project 내의 모든 URL 패턴을 바꿀 수 있도록 도와줍니다.

request 와 response 의 기본 흐름을 이해하셨다면, 튜토리얼 2장 에서 데이터베이스 작업을 시작해보세요.

‹ 빠른 설치 가이드 첫 번째 장고 앱 작성하기, part 2 ›

BACK TO TOP

목차

Browse

- 이전: 빠른 설치 가이드
- 다음: 첫 번째 장고 앱 작성하기, part 2
- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 1

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list
|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 2

이 튜토리얼은 튜토리얼 1장에서 이어집니다. 데이터베이스를 설치하고 첫 모델을 생성한 후, Django에서 자동 생성되는 관리자 사이트에 대해 짧게 소개합니다.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

데이터베이스 설치

이제, **mysite/settings.py** 파일을 열어보세요. 이 파일은 Django 설정을 모듈 변수로 표현한 보통의 Python 모듈입니다.

기본적으로는 SQLite을 사용하도록 구성되어 있습니다. 만약 데이터베이스를 처음 경험해보거나, Django에서 데이터베이스를 한 번 경험해 보고 싶다면, SQLite가 가장 간단한 방법입니다. SQLite는 Python에서 기본으로 제공되기 때문에 별도로 설치할 필요가 없습니다. 그러나 실제 프로젝트를 시작할 때에는, 나중에 데이터베이스를 교체하느라 골치 아파질 일을 피하기 위해서라도 PostgreSQL 같이 좀 더 확장성 있는 데이터베이스를 사용하는 것이 좋습니다.

다른 데이터베이스를 사용해보고 싶다면, 적절한 데이터베이스 바인딩을 설치하고, 데이터베이스 연결 설정과 맞게끔 **DATABASES** **'default'** 항목의 값을 다음의 키 값으로 바꿔주세요.

- **ENGINE** – **'django.db.backends.sqlite3'**, **'django.db.backends.postgresql'**, **'django.db.backends.mysql'**, 또는 **'django.db.backends.oracle'**. 그외에 서드파티 백엔드 참조.
- **NAME** – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, **NAME** should be the full absolute path, including filename, of that file. The default value, **BASE_DIR / 'db.sqlite3'**, will store the file in your project directory.

SQLite 를 데이터베이스로 사용하지 않는 경우, **USER**, **PASSWORD**, **HOST** 같은 추가 설정이 반드시 필요합니다. 더 자세한 내용은 **DATABASES** 문서를 참조해 주세요.

SQLite 이외의 데이터베이스라면

만약 SQLite 이외의 데이터베이스를 사용하는 경우, 이 시점에서 데이터베이스를 생성해야 합니다. 데이터베이스의 대화형 프롬프트 내에서 《**CREATE DATABASE database_name;**》 명령을 실행하면 됩니다.

또한, **mysite/settings.py** 에 설정된 데이터베이스 사용자가 《create database》 권한이 있는지도 확인해 봐야 합니다. 튜토리얼을 진행하며 필요한 경우 테스트 데이터베이스를 자동으로 생성할 수 있도록 해줍니다.

SQLite를 사용한다면 아무것도 미리 생성할 필요가 없습니다. 데이터베이스 파일은 필요할 때마다 자동으로 생성됩니다.

mysite/settings.py를 편집할 때, 당신의 시간대에 맞춰 **TIME_ZONE** 값을 설정하세요.

또한, 이 파일의 윗쪽에 있는 **INSTALLED_APPS** 에 대해 언급하자면, 이 파일은 현재 Django 인스턴스에서 활성화된 모든 Django 어플리케이션들의 이름이 담겨 있습니다. 앱들은 다수의 프로젝트에서 사용될 수 있고, 다른 프로젝트에서 쉽게 사용될 수 있도록

패키징하여 배포할 수 있습니다.

기본적으로는, **`INSTALLED_APPS`**는 Django와 함께 딸려오는 다음의 앱들을 포함합니다.

- **`django.contrib.admin`** – 관리용 사이트. 곧 사용하게 될 겁니다.
- **`django.contrib.auth`** – 인증 시스템.
- **`django.contrib.contenttypes`** – 컨텐츠 타입을 위한 프레임워크.
- **`django.contrib.sessions`** – 세션 프레임워크.
- **`django.contrib.messages`** – 메세징 프레임워크.
- **`django.contrib.staticfiles`** – 정적 파일을 관리하는 프레임워크.

이 어플리케이션들은 일반적인 경우에 사용하기 편리하도록 기본으로 제공됩니다.

이러한 기본 어플리케이션들 중 몇몇은 최소한 하나 이상의 데이터베이스 테이블을 사용하는데, 그러기 위해서는 데이터베이스에서 테이블을 미리 만들 필요가 있습니다. 이를 위해, 다음의 명령을 실행해봅시다.

```
$ python manage.py migrate
```

The **`migrate`** command looks at the **`INSTALLED_APPS`** setting and creates any necessary database tables according to the database settings in your **`mysite/settings.py`** file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies. If you're interested, run the command-line client for your database and type **`\dt`** (PostgreSQL), **`SHOW TABLES;`** (MariaDB, MySQL), **`.schema`** (SQLite), or **`SELECT TABLE_NAME FROM USER_TABLES;`** (Oracle) to display the tables Django created.

> **최소주의자(minimalists)들을 위하여**
>
> 위에서 언급했다시피, 기본으로 제공되는 어플리케이션은 일반적인 상황을 염두에 두었으나, 모두에게 필요한 것은 아닙니다. 만약 이것들이 필요 없다고 생각되시면, **`migrate`**를 실행하기 전에 **`INSTALLED_APPS`**에서 제거할 어플리케이션들을 그냥 주석처리(comment-out) 하시거나 삭제하시면 됩니다. **`migrate`** 명령은 **`INSTALLED_APPS`**에 등록된 어플리케이션에 한하여 실행될 것입니다.

모델 만들기

이제, 모델을 정의해 보겠습니다. 본질적으로, 모델이란 부가적인 메타데이터를 가진 데이터베이스의 구조(layout)를 말합니다.

> **철학**
>
> 모델(《model》)은 데이터에 관한 단 하나의, 가장 확실한 진리의 원천입니다. 이것은 당신이 저장하는 데이터의 필수적인 필드들과 동작들을 포함하고 있습니다. Django 는 DRY 원칙 을 따릅니다. 이 원칙에 따라 데이터 모델을 한곳에서 정의하고, 이것으로부터 자동으로 뭔가를 유도하는 것이 목표입니다.
>
> This includes the migrations - unlike in Ruby On Rails, for example, migrations are entirely derived from your models file, and are essentially a history that Django can roll through to update your database schema to match your current models.

In our poll app, we'll create two models: **`Question`** and **`Choice`**. A **`Question`** has a question and a publication date. A **`Choice`** has two fields: the text of the choice and a vote tally. Each **`Choice`** is associated with a **`Question`**.

These concepts are represented by Python classes. Edit the **`polls/models.py`** file so it looks like this:

```
polls/models.py
```

```python
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Here, each model is represented by a class that subclasses **`django.db.models.Model`**. Each model has a number of class variables, each of which represents a database field in the model.

데이터베이스의 각 필드는 **`Field`** 클래스의 인스턴스로서 표현됩니다. **`CharField`** 는 문자(character) 필드를 표현하고, **`DateTimeField`** 는 날짜와 시간(datetime) 필드를 표현합니다. 이것은 각 필드가 어떤 자료형을 가질 수 있는지를 Django 에게 말해줍니다.

각각의 **`Field`** 인스턴스의 이름(**`question_text`** 또는 **`pub_date`**)은 기계가 읽기 좋은 형식(machine-friendly format)의 데이터베이스 필드 이름입니다. 이 필드명을 Python 코드에서 사용할 수 있으며, 데이터베이스에서는 컬럼명으로 사용할 것입니다.

`Field` 클래스의 생성자에 선택적인 첫번째 위치 인수를 전달하여 사람이 읽기 좋은(human-readable) 이름을 지정할 수도 있습니다. 이 방법은 Django 의 내부를 설명하는 용도로 종종 사용되는데, 이는 마치 문서가 늘어나는 것 같은 효과를 가집니다. 만약 이 선택적인 첫번째 위치 인수를 사용하지 않으면, Django 는 기계가 읽기 좋은 형식의 이름을 사용합니다. 이 예제에서는, **`Question.pub_date`** 에 한해서만 인간이 읽기 좋은 형태의 이름을 정의하겠습니다. 그 외의 다른 필드들은, 기계가 읽기 좋은 형태의 이름이라도 사람이 읽기에는 충분합니다.

몇몇 **`Field`** 클래스들은 필수 인수가 필요합니다. 예를 들어, **`CharField`** 의 경우 **`max_length`** 를 입력해 주어야 합니다. 이것은 데이터베이스 스키마에서만 필요한것이 아닌 값을 검증할때도 쓰이는데, 곧 보게 될것입니다.

또한 **`Field`** 는 다양한 선택적 인수들을 가질 수 있습니다. 이 예제에서는, **`default`** 로 하여금 **`votes`** 의 기본값을 0 으로 설정하였습니다.

마지막으로, **`ForeignKey`** 를 사용한 관계설정에 대해 설명하겠습니다. 이 예제에서는 각각의 **`Choice`** 가 하나의 **`Question`** 에 관계된다는 것을 Django 에게 알려줍니다. Django 는 다-대-일(many-to-one), 다-대-다(many-to-many), 일-대-일(one-to-one) 과 같은 모든 일반 데이터베이스의 관계들를 지원합니다.

모델의 활성화

모델에 대한 이 작은 코드가, Django에게는 상당한 양의 정보를 전달합니다. Django는 이 정보를 가지고 다음과 같은 일을 할 수 있습니다.

- 이 앱을 위한 데이터베이스 스키마 생성(**`CREATE TABLE`** 문)
- **`Question`**과 **`Choice`** 객체에 접근하기 위한 Python 데이터베이스 접근 API를 생성

그러나, 가장 먼저 현재 프로젝트에게 **`polls`** 앱이 설치되어 있다는 것을 알려야 합니다.

철학

Django의 앱들은 《꼈다뺐다》 할 수 있습니다. 앱을 다수의 프로젝트에서 사용할 수 있으며, 앱을 배포할 수도 있습니다. 특정 Django 사이트에 앱들이 묶여있지 않아도 되기 때문입니다.

앱을 현재의 프로젝트에 포함시키기 위해서는, 앱의 구성 클래스에 대한 참조를 **`INSTALLED_APPS`** 설정에 추가해야 합니다. **`PollsConfig`** 클래스는 **`polls/apps.py`** 파일 내에 존재합니다. 따라서, 점으로 구분된 경로는 **`'polls.apps.PollsConfig'`**가 됩니다. 이 점으로 구분된 경로를, **`mysite/settings.py`** 파일을 편집하여

INSTALLED_APPS 설정에 추가하면 됩니다. 이는 다음과 같이 보일 것입니다.

mysite/settings.py

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

이제, Django는 **polls** 앱이 포함된 것을 알게 되었습니다. 다른 명령을 내려봅시다.

```
$ python manage.py makemigrations polls
```

다음과 비슷한 것이 보일 겁니다.

```
Migrations for 'polls':
  polls/migrations/0001_initial.py
    - Create model Question
    - Create model Choice
```

makemigrations 을 실행시킴으로서, 당신이 모델을 변경시킨 사실과(이 경우에는 새로운 모델을 만들었습니다) 이 변경사항을 *migration*으로 저장시키고 싶다는 것을 Django에게 알려줍니다.

Migrations are how Django stores changes to your models (and thus your database schema) - they're files on disk. You can read the migration for your new model if you like; it's the file **polls/migrations/0001_initial.py**. Don't worry, you're not expected to read them every time Django makes one, but they're designed to be human-editable in case you want to manually tweak how Django changes things.

당신을 위해 migration들을 실행시켜주고, 자동으로 데이터베이스 스키마를 관리해주는 **migrate** 명령어가 있습니다. 이 명령을 알아보기 전에 migration이 내부적으로 어떤 SQL 문장을 실행하는지 살펴봅시다. **sqlmigrate** 명령은 migration 이름을 인수로 받아, 실행하는 SQL 문장을 보여줍니다.

```
$ python manage.py sqlmigrate polls 0001
```

다음과 비슷한 결과를 보실 수 있습니다. (가독성을 위해 결과물을 조금 다듬었습니다)

```
BEGIN;
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL,
    "question_id" integer NOT NULL
);
ALTER TABLE "polls_choice"
  ADD CONSTRAINT "polls_choice_question_id_c5b4b260_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice" ("question_id");

COMMIT;
```

다음을 참고하세요.

- 사용하는 데이터베이스에 따라서 출력결과는 다를 수 있습니다. 위의 출력결과는 PostgreSQL에서 생성되었습니다.
- 테이블 이름은 앱의 이름과 모델의 이름(소문자)이 조합되어 자동으로 생성됩니다. 이 경우, 앱의 이름인 **polls** 와 소문자로 표기된 모델의 이름인 **question** 과 **choice**가 합쳐집니다. (이 동작을 재지정(override)하여 수정할 수 있습니다.)
- 기본 키(ID)가 자동으로 추가됩니다. (역시 이 동작도 재지정할 수 있습니다.)
- 관례에 따라, Django는 외래 키 필드명에 **"_id"** 이름을 자동으로 추가합니다. (물론 이것도 재지정할 수 있습니다.)
- The foreign key relationship is made explicit by a **FOREIGN KEY** constraint. Don't worry about the **DEFERRABLE** parts; it's telling PostgreSQL to not enforce the foreign key until the end of the transaction.
- 사용하는 데이터베이스에 따라, 데이터베이스 고유의 필드타입이 조정됩니다. 따라서, 자동 증가 필드를 생성할 경우 **auto_increment**(MySQL), **serial**(PostgreSQL), **integer primary key autoincrement**(SQLite)와 같이 사용하는 데이터베이스에 따라 적절한 필드타입이 자동으로 선택됩니다. 필드 명에 사용되는 인용부호도 상황에 따라 겹따옴표나 홑따옴표가 적절히 선택됩니다.
- The **sqlmigrate** command doesn't actually run the migration on your database - instead, it prints it to the screen so that you can see what SQL Django thinks is required. It's useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

관심이 있다면, **python manage.py check** 명령을 통해 마이그레이션을 수행하거나 데이터베이스를 건드리지 않고도 프로젝트의 문제를 확인할 수 있습니다.

이제, **migrate** 를 실행시켜 데이터베이스에 모델과 관련된 테이블을 생성해봅시다.

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

migrate 명령은 아직 적용되지 않은 마이그레이션을 모두 수집해 이를 실행하며(Django는 **django_migrations** 테이블을 두어 마이그레이션 적용 여부를 추적합니다) 이 과정을 통해 모델에서의 변경 사항들과 데이터베이스의 스키마의 동기화가 이루어집니다.

마이그레이션은 매우 기능이 강력하여, 마치 프로젝트를 개발할 때처럼 데이터베이스나 테이블에 손대지 않고도 모델의 반복적인 변경을 가능하게 해줍니다. 동작 중인 데이터베이스를 자료 손실 없이 업그레이드 하는 데 최적화 되어 있습니다. 튜토리얼의 나머지 부분에서 이 부분을 조금 더 살펴보겠습니다만, 지금은 모델의 변경을 만드는 세 단계의 지침을 기억하세요.

- (**models.py** 에서) 모델을 변경합니다.
- **python manage.py makemigrations**을 통해 이 변경사항에 대한 마이그레이션을 만드세요.
- **python manage.py migrate** 명령을 통해 변경사항을 데이터베이스에 적용하세요.

마이그레이션을 만드는 명령과 적용하는 명령이 분리된 것은 버전 관리 시스템에 마이그레이션을 커밋하고 앱과 함께 출시할 수 있도록 하기 위해서입니다. 이는 당신의 개발을 쉽게 해줄 뿐 아니라, 다른 개발자가 프로덕션에서 사용할 수 있게 해줍니다.

manage.py 유틸리티로 어떤 일들을 할 수 있는지 django-admin 문서를 읽어보세요.

API 가지고 놀기

이제, 대화식 Python 쉘에 뛰어들어 Django API를 자유롭게 가지고 놀아봅시다. Python 쉘을 실행하려면 다음의 명령을 입력합니다.

```
$ python manage.py shell
```

We're using this instead of simply typing 《python》, because **manage.py** sets the **DJANGO_SETTINGS_MODULE** environment variable, which gives Django the Python import path to your **mysite/settings.py** file.

쉘에 진입한 후, 데이터베이스 API를 탐험해 보세요.

```
>>> from polls.models import Choice, Question  # Import the model classes we just wrote.

# No questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

여기서 잠깐. **<Question: Question object (1)>**은 이 객체를 표현하는 데 별로 도움이 되지 않습니다.
(**polls/models.py** 파일의) **Question** 모델을 수정하여, **__str__()** 메소드를 **Question**과 **Choice**에 추가해 봅시다.

polls/models.py

```
from django.db import models

class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

당신의 모델에 **__str__()** 메소드를 추가하는것은 객체의 표현을 대화식 프롬프트에서 편하게 보려는 이유 말고도, Django 가 자동으로 생성하는 관리 사이트 에서도 객체의 표현이 사용되기 때문입니다.

이 모델에 커스텀 메소드 또한 추가해봅시다:

polls/models.py

```
import datetime

from django.db import models
from django.utils import timezone


class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

import datetime은 Python의 표준 모듈인 **datetime** 모듈을, **from django.utils import timezone**은 Django의 시간대 관련 유틸리티인 **django.utils.timezone**을 참조하기 위해 추가한 것입니다. 만약 Python에서 시간대를 조작하는 방법에 대해 익숙하지 않다면, 시간대 지원 문서에서 더 많은 것을 배울 수 있습니다.

변경된 사항을 저장하고, **python manage.py shell**를 다시 실행해보세요.

```
>>> from polls.models import Choice, Question

# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
```

```
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

모델의 관계에 대한 더 많은 정보는 관련 객체에 접근하기를 참조하세요. API에서 이중 밑줄(__) 을 이용해서 어떻게 필드를 조회할 수 있는지는 필드 조회를 읽어보세요.데이터베이스 API에 대한 자세한 내용을 보려면, 데이터베이스 API 레퍼런스를 읽어보세요.

Django 관리자 소개

철학

직원들이나 고객들이 컨텐츠를 수정하기 위한 관리자 사이트를 만드는 것은 딱히 창의적일 필요없는 지루한 작업입니다. 이런 이유로, Django는 모델에 대한 관리용 인터페이스를 모두 자동으로 생성합니다.

Django는 **Lawrence Journal-World** 신문사의 프로그래머가 처음 개발하였습니다. 이런 태생적인 이유 때문에, 《컨텐츠 게시자》 와 《공개》 사이트의 구분이 명확합니다. 사이트 관리자는 뉴스 기사, 사건, 스포츠 경기 결과 같은 것들을 시스템에 추가합니다. 그렇게 추가된 컨텐츠는 《공개》 사이트에 노출됩니다. Django는 사이트 관리자가 컨텐츠를 편집할 수 있는 통합적인 인터페이스를 생성하는 문제를 해결합니다.

관리자 사이트는 사이트 방문자를 위한 것이 아니라, 사이트 관리자를 위한 것입니다.

관리자 생성하기

우선, 관리 사이트에 로그인 할 수 있는 사용자를 생성해 봅시다. 다음과 같은 명령을 수행합니다.

```
$ python manage.py createsuperuser
```

원하는 username 을 입력하고 엔터를 누르세요

```
Username: admin
```

그런 다음 원하는 이메일 주소를 입력하라는 메시지가 표시됩니다.

```
Email address: admin@example.com
```

마지막으로, 암호를 입력하세요. 암호를 두번 물어보게 되는데, 두번째 입력하는 암호를 올바로 입력했는지를 확인하기 위한 암호입니다.

```
Password: **********
Password (again): *********
Superuser created successfully.
```

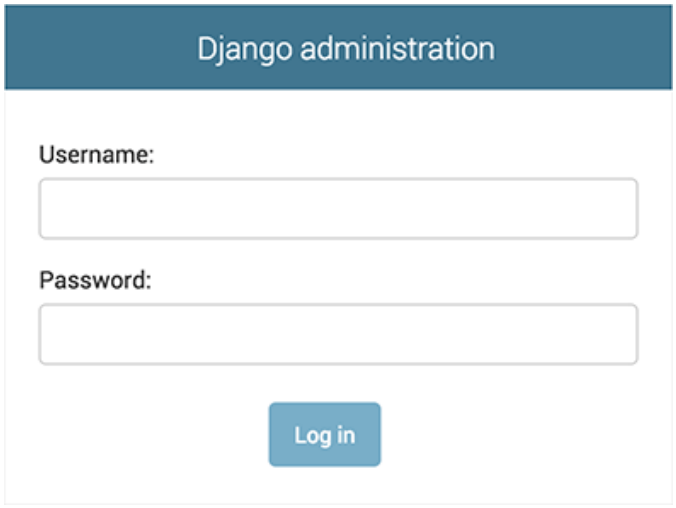
개발 서버 시작

Django 관리자 사이트는 기본으로 활성화되어 있습니다. 개발 서버를 켜고, 탐험해 봅시다.

서버가 동작하고 있지 않다면 다음 명령으로 시작합니다.

```
$ python manage.py runserver
```

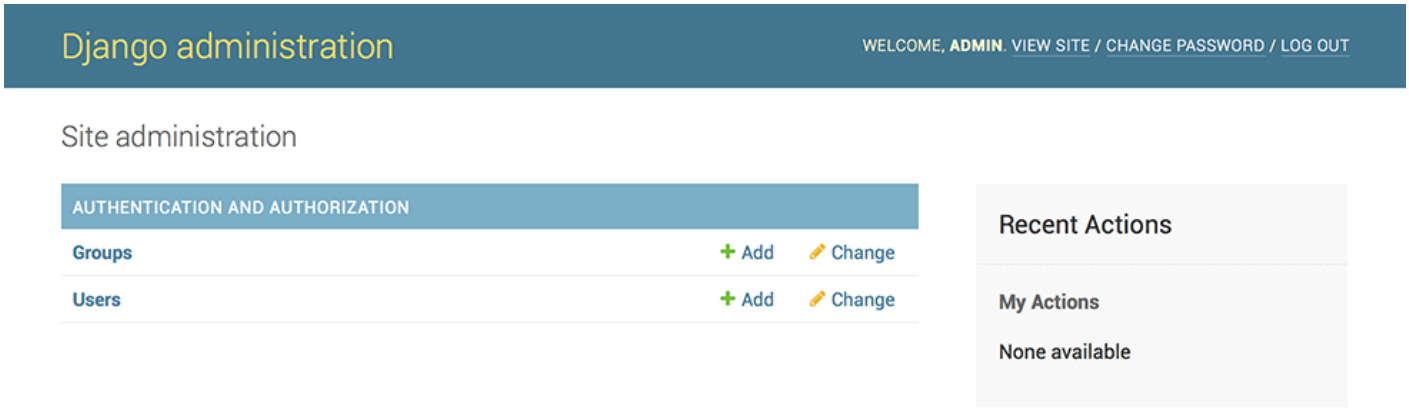
이제, 웹 브라우저를 열고 로컬 도메인의 《/admin/》으로 이동합니다. 예를 들면, http://127.0.0.1:8000/admin/ 으로 접근할 수 있습니다. 그럼 다음과 같이 로그인 화면이 보일 겁니다.



Since translation is turned on by default, if you set **LANGUAGE_CODE**, the login screen will be displayed in the given language (if Django has appropriate translations).

관리자 사이트에 들어가기

앞서 생성한 슈퍼유저(superuser) 계정으로 로그인해봅시다. 다음과 같은 Django 관리 인덱스 페이지가 보일 것입니다.



편집 가능한 그룹과 사용자와 같은 몇 종류의 컨텐츠를 볼 수 있습니다. 이것들은 **`django.contrib.auth`** 모듈에서 제공되는데, Django 에서 제공되는 인증 프레임워크 입니다.

관리 사이트에서 poll app 을 변경가능하도록 만들기

그런데, poll app 이 관리 인덱스 페이지에서 보이지 않네요. 어디에 있을까요?

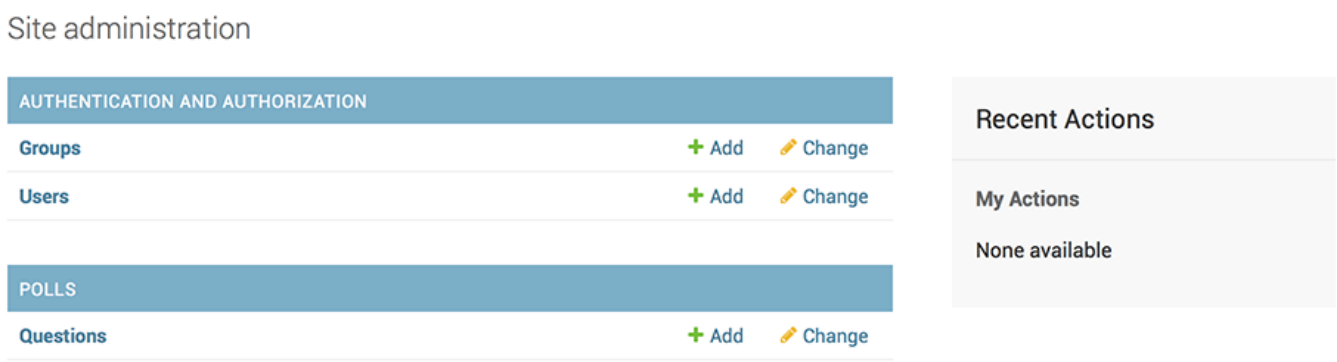
Only one more thing to do: we need to tell the admin that **`Question`** objects have an admin interface. To do this, open the **`polls/admin.py`** file, and edit it to look like this:

polls/admin.py

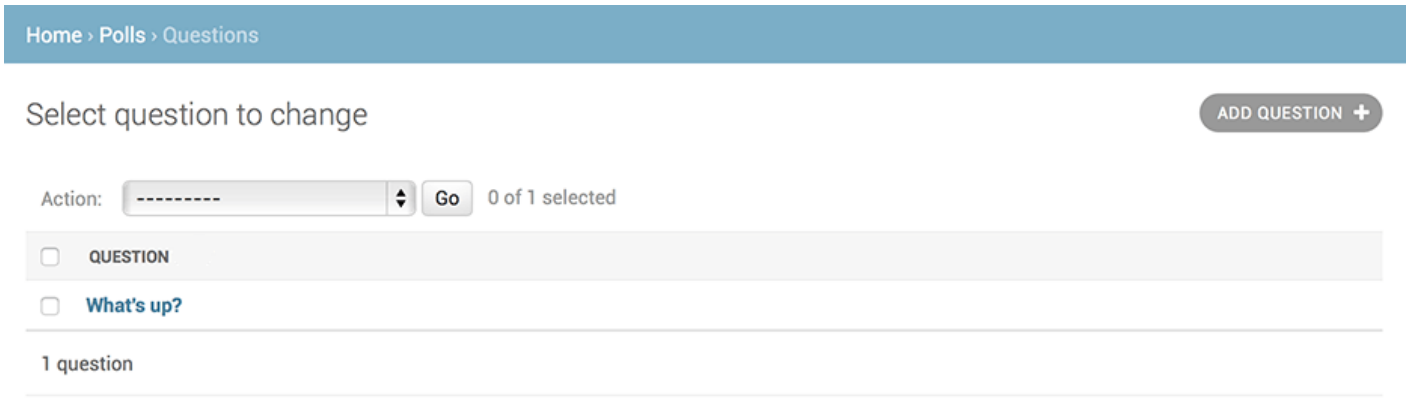
```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

자유로운 관리 기능을 탐색하기

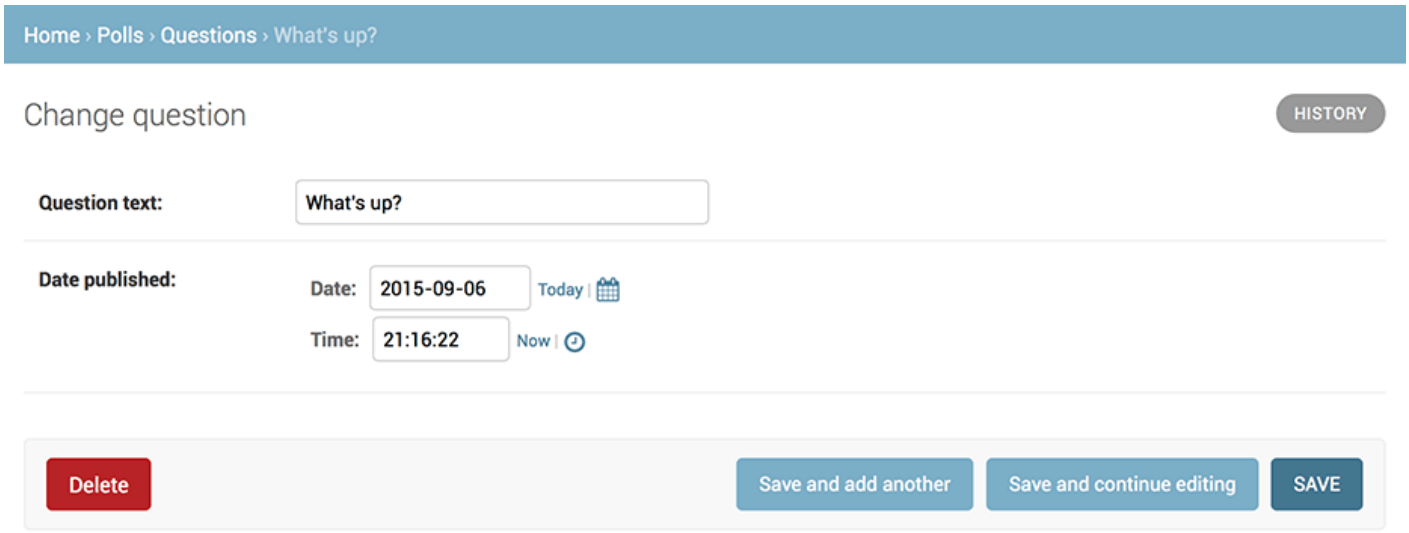
이제, **`Question`** 을 등록시켰으니 Django 는 이를 알아채고 관리 인덱스 페이지에 이를 표시할 것입니다:



《Questions》 을 클릭하십시오. 그러면 질문들을 위한 《change list》 로 이동합니다. 이 페이지는 데이터베이스에 저장된 모든 질문들을 보여주며, 그 중 하나를 선택하여 변경할 수 있습니다. 이전에 등록했던 《What's up?》 질문이 있을 것입니다.



《What's up?》 질문을 클릭하여 수정합니다.



여기서 알아둘 것들:

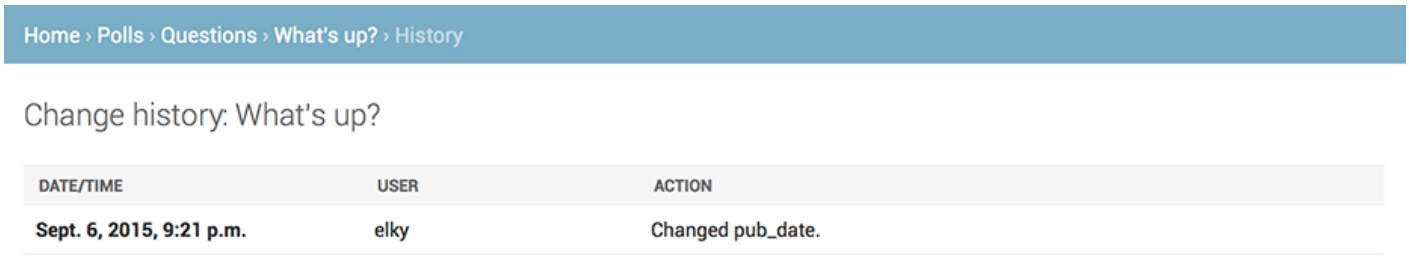
- 이 서식은 **`Question`** 모델에서 자동으로 생성되었습니다
- 모델의 각 필드 유형들은 (**`DateTimeField`**, **`CharField`**) 적절한 HTML 입력 위젯으로 표현됩니다. 필드의 각 유형들은 Django 관리 사이트에서 어떻게 표현해되어야 할지 알고 있습니다.
- 각각의 **`DateTimeField`** 는 JavaScript 로 작성된 단축 기능과 연결됩니다. 날짜는 《오늘》(《Today》) 버튼과 달력 팝업에서 입력할 수 있으며, 시간은 《지금》(《Now》) 버튼과 일반적으로 입력하는 시간들을 제공하는 편리한 팝업을 통해서도 입력할 수 있습니다.

페이지의 아래 부분에서 다음과 같은 몇가지 옵션을 제공합니다.

- 저장(Save) – 이 유형의 객체에 대한 변경사항을 저장하고, 변경된 목록 페이지를 보여줍니다
- 저장 및 편집 계속(Save and continue editing) – 이 객체에 대한 변경사항을 저장하고, 현재 편집창을 갱신합니다
- 저장 및 다른 이름으로 추가(Save and add another) – 변경사항을 저장하고, 이 유형의 객체에 대한 비어있는 새로운 입력창을 불러옵니다
- 삭제(Delete) – 삭제를 확인하는 페이지를 띄웁니다.

만약 《Date published》 의 값이 Tutorial 1 에서 질문을 생성했을때의 시간과 일치하지 않는다면, **`TIME_ZONE`** (시간대) 설정을 깜빡 하신것일지도 모릅니다. 이 설정을 바꾸시고 다시 페이지를 불러오시면 올바른 값이 표현됩니다.

《Date published》 의 값을 《오늘》(《Today》) 과 《지금》(《Now》) 단축버튼을 눌러 바꾸십시요. 그런 후, 《저장 및 편집 계속》(《Save and continue editing》) 을 누르십시요. 그런 후, 우측 상단의 《히스토리》(《History》) 버튼을 누르십시요. Django 관리사이트를 통해 누가(username) 언제(timestamp) 무엇을 바꾸었는지 목록을 확인할 수 있습니다.



모델 API와 관리 사이트에 익숙해졌다면, 이 튜토리얼의 3장에서 투표 앱에 뷰를 추가하는 방법을 배워보세요.

‹ 첫 번째 장고 앱 작성하기, part 1

첫 번째 장고 앱 작성하기, part 3 ›

^ BACK TO TOP

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 1
- 다음: 첫 번째 장고 앱 작성하기, part 3
- 목차

- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 2

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list
|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 3

이 튜토리얼은 튜토리얼 2장 에서 이어집니다. 이제 투표 어플리케이션에 공개 인터페이스인 《뷰(view)》를 추가해 보겠습니다.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

개요

뷰는 Django 어플리케이션이 일반적으로 특정 기능과 템플릿을 제공하는 웹페이지의 한 종류입니다. 예를 들어, 블로그 어플리케이션의 경우 다음과 같은 뷰를 가질 수 있습니다.

- Blog 홈페이지 – 가장 최근의 항목들을 보여줍니다
- 항목 《세부》(detail) 페이지 – 하나의 항목에 연결하는 영구적인 링크(permalink)를 제공합니다.
- 년도별 축적 페이지 – 주어진 연도의 모든 월별 항목들을 표시합니다.
- 월별 축적 페이지 – 주어진 월의 날짜별 항목들을 표시합니다.
- 날짜별 축적 페이지 – 주어진 날짜의 모든 항목들을 표시합니다.
- 댓글 기능 – 특정 항목의 댓글을 다룰 수 있는 기능

우리가 만드는 poll 어플리케이션에서 다음과 같은 네개의 view 를 만들어 보겠습니다.

- 질문 《색인》 페이지 – 최근의 질문들을 표시합니다.
- 질문 《세부》 페이지 – 질문 내용과, 투표할 수 있는 서식을 표시합니다.
- 질문 《결과》 페이지 – 특정 질문에 대한 결과를 표시합니다
- 투표 기능 – 특정 질문에 대해 특정 선택을 할 수 있는 투표 기능을 제공합니다.

In Django, web pages and other content are delivered by views. Each view is represented by a Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that's requested (to be precise, the part of the URL after the domain name).

Now in your time on the web you may have come across such beauties as **`ME2/Sites/dirmod.htm?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B`**. You will be pleased to know that Django allows us much more elegant *URL patterns* than that.

A URL pattern is the general form of a URL - for example: **`/newsarchive/<year>/<month>/`**.

URL로부터 뷰를 얻기 위해, Django는 〈URLconfs'라는 것을 사용합니다. URLconf는 URL 패턴을 뷰에 연결합니다.

이 튜토리얼은 URLconfs를 사용하는 기초 지식을 제공하며, 좀 더 자세한 정보는 URL dispatcher를 참조하세요.

뷰 추가하기

이제, **polls/views.py** 에 뷰를 추가해 봅시다. 이 뷰들은 인수를 받기 때문에 조금 모양이 다릅니다.

polls/views.py

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

다음의 **path()** 호출을 추가하여 이러한 새로운 뷰를 **polls.urls** 모듈로 연결하세요.

polls/urls.py

```
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

브라우저에 《/polls/34/》 를 입력해 보세요. 이 주소에 접속하면 **detail()** 함수를 호출하여 URL 에 입력한 ID 를 출력할 것입니다. 《/polls/34/results/》 와 《/polls/34/vote/》 도 실행해 보세요. 투표 결과와 투표 페이지의 뼈대가 되는 페이지가 출력될 것입니다.

사용자가 웹사이트의 페이지를 요청할 때, 예로 《/polls/34/》를 요청했다고 하면, Django는 **mysite.urls** 파이썬 모듈을 불러오게 됩니다. **ROOT_URLCONF** 설정에 의해 해당 모듈을 바라보도록 지정되어 있기 때문입니다. **mysite.urls**에서 **urlpatterns**라는 변수를 찾고, 순서대로 패턴을 따라갑니다. **'polls/'**를 찾은 후엔, 일치하는 텍스트(**"polls/"**)를 버리고, 남은 텍스트인 **"34/"**를 〈polls.urls〉 URLconf로 전달하여 남은 처리를 진행합니다. 거기에 **'<int:question_id>/'**와 일치하여, 결과적으로 **detail()** 뷰 함수가 호출됩니다.

```
detail(request=<HttpRequest object>, question_id=34)
```

question_id=34 부분은 **<int:question_id>** 에서 왔습니다. 괄호를 사용하여 URL 의 일부를 《캡처》 하고, 해당 내용을 keyword 인수로서 뷰 함수로 전달합니다. 문자열의 **:question_id>** 부분은 일치되는 패턴을 구별하기 위해 정의한 이름이며, **<int:** 부분은 어느 패턴이 해당 URL 경로에 일치되어야 하는 지를 결정하는 컨버터입니다.

뷰가 실제로 뭔가를 하도록 만들기

각 뷰는 두 가지 중 하나를 하도록 되어 있습니다. 요청된 페이지의 내용이 담긴 **HttpResponse** 객체를 반환하거나, 혹은 **Http404** 같은 예외를 발생하게 해야합니다. 나머지는 당신에게 달렸습니다.

당신이 작성한 뷰는 데이터베이스의 레코드를 읽을 수도 있습니다. 또한 뷰는 Django나 Python에서 서드파티로 제공되는 템플릿 시스템을 사용할 수도 있습니다. 뷰는 PDF를 생성하거나, XML을 출력하거나, 실시간으로 ZIP 파일을 만들 수 있습니다. 뷰는 당신이 원하는 무엇이든, Python의 어떤 라이브러리라도 사용할 수 있습니다.

Django에 필요한 것은 **HttpResponse** 객체 혹은 예외입니다.

왜냐면, 그렇게 다루는게 편리하기 때문입니다. 튜토리얼 2장의 예제에서 다룬 Django 자체 데이터베이스 API를 사용해봅시다. 새로운 **index()** 뷰 하나를 호출했을 때, 시스템에 저장된 최소한 5 개의 투표 질문이 콤마로 분리되어, 발행일에 따라 출력됩니다.

polls/views.py

```python
from django.http import HttpResponse

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged
```

여기 몇가지 문제가 있습니다. 뷰에서 페이지의 디자인이 하드코딩 되어 있다고 합시다. 만약 페이지가 보여지는 방식을 바꾸고 싶다면, 이 Python 코드를 편집해야만 할 겁니다. 그럼, 뷰에서 사용할 수 있는 템플릿을 작성하여, Python 코드로부터 디자인을 분리하도록 Django의 템플릿 시스템을 사용해 봅시다.

우선, **polls** 디렉토리에 **templates**라는 디렉토리를 만듭니다. Django는 여기서 템플릿을 찾게 될 것입니다.

프로젝트의 **TEMPLATES** 설정은 Django가 어떻게 템플릿을 불러오고 렌더링 할 것인지 기술합니다. 기본 설정 파일은 **APP_DIRS** 옵션이 **True**로 설정된 **DjangoTemplates** 백엔드를 구성합니다. 관례에 따라, **DjangoTemplates**은 각 **INSTALLED_APPS** 디렉토리의 《templates》 하위 디렉토리를 탐색합니다.

Within the **templates** directory you have just created, create another directory called **polls**, and within that create a file called **index.html**. In other words, your template should be at **polls/templates/polls/index.html**. Because of how the **app_directories** template loader works as described above, you can refer to this template within Django as **polls/index.html**.

템플릿 네임스페이싱

Now we *might* be able to get away with putting our templates directly in **polls/templates** (rather than creating another **polls** subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

템플릿에 다음과 같은 코드를 입력합니다.

polls/templates/polls/index.html

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

주석

To make the tutorial shorter, all template examples use incomplete HTML. In your own projects you should use complete HTML documents.

이제, 템플릿을 이용하여 **polls/views.py**에 **index** 뷰를 업데이트 해보도록 하겠습니다.

polls/views.py

```
from django.http import HttpResponse
from django.template import loader

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

이 코드는 **polls/index.html** 템플릿을 불러온 후, context를 전달합니다. context는 템플릿에서 쓰이는 변수명과 Python 객체를 연결하는 사전형 값입니다.

브라우저에서 《/polls/》 페이지를 불러오면, 튜토리얼 2장에서 작성한 《What's up》 질문이 포함된 리스트가 표시됩니다. 표시된 질문의 링크는 해당 질문에 대한 세부 페이지를 가리킵니다.

지름길: `render()`

템플릿에 context 를 채워넣어 표현한 결과를 **HttpResponse** 객체와 함께 돌려주는 구문은 자주 쓰는 용법입니다. 따라서 Django는 이런 표현을 쉽게 표현할 수 있도록 단축 기능(shortcuts)을 제공합니다. **index()** 뷰를 단축 기능으로 작성하면 다음과 같습니다.

polls/views.py

```
from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

모든 뷰에 적용한다면, 더 이상 **loader**와 **HttpResponse**를 임포트하지 않아도 됩니다. (만약 **detail**, **results**, **vote**에서 stub 메소드를 가지고 있다면, **HttpResponse**를 유지해야 할 것입니다.)

render() 함수는 request 객체를 첫번째 인수로 받고, 템플릿 이름을 두번째 인수로 받으며, context 사전형 객체를 세전째 선택적(optional) 인수로 받습니다. 인수로 지정된 context로 표현된 템플릿의 **HttpResponse** 객체가 반환됩니다.

---

404 에러 일으키기

이제, 질문의 상세 뷰에 태클을 걸어보겠습니다. 상세 뷰는 지정된 설문조사의 질문 내용을 보여줍니다. 다음과 같습니다.

polls/views.py

```
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

여기 새로운 내용이 추가되었습니다. 뷰는 요청된 질문의 ID 가 없을 경우 **Http404** 예외를 발생시킵니다.

조금 후에 **polls/detail.html** 템플릿에 무엇을 넣을 수 있는지 논의하겠지만, 일단 위의 예제를 동작시키기 위해 아래의 내용이 들어있는 파일을 작성하세요.

polls/templates/polls/detail.html

```
{{ question }}
```

이제 시작해도 됩니다.

지름길: get_object_or_404()

만약 객체가 존재하지 않을 때 **get()** 을 사용하여 **Http404** 예외를 발생시키는것은 자주 쓰이는 용법입니다. Django에서 이 기능에 대한 단축 기능을 제공합니다. **detail()** 뷰를 단축 기능으로 작성하면 다음과 같습니다.

polls/views.py

```python
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

get_object_or_404() 함수는 Django 모델을 첫번째 인자로 받고, 몇개의 키워드 인수를 모델 관리자의 **get()** 함수에 넘깁니다. 만약 객체가 존재하지 않을 경우, **Http404** 예외가 발생합니다.

철학

상위 계층에서 **ObjectDoesNotExist** 예외를 자동으로 잡아 내는 대신 **get_object_or_404()** 도움 함수(helper functoin)를 사용하거나, **ObjectDoesNotExist** 예외를 사용하는 대신 **Http404** 를 사용하는 이유는 무엇일까요?

왜냐하면, 모델 계층을 뷰 계층에 연결하는 방법이기 때문입니다. Django의 중요한 설계 목표는, 약결합(loose coupling)을 관리하는 데에 있습니다. 일부 제어된 결합이 **django.shortcuts** 모듈에서 도입되었습니다.

또한, **get_object_or_404()** 함수처럼 동작하는 **get_list_or_404()** 함수가 있습니다. **get()** 대신 **filter()** 를 쓴다는 것이 다릅니다. 리스트가 비어있을 경우, **Http404** 예외를 발생시킵니다.

템플릿 시스템 사용하기

투표 어플리케이션의 **detail()** 뷰로 되돌아가 봅시다. context 변수 **question**이 주어졌을때, **polls/detail.html**이라는 템플릿이 어떻게 보이는지 봅시다.

polls/templates/polls/detail.html 

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

템플릿 시스템은 변수의 속성에 접근하기 위해 점-탐색(dot-lookup) 문법을 사용합니다. 예제의 **{{ question.question_text }}** 구문을 보면, Django는 먼저 **question** 객체에 대해 사전형으로 탐색합니다. 탐색에 실패하게 되면 속성값으로 탐색합니다. (이 예에서는 속성값에서 탐색이 완료됩니다만) 만약 속성 탐색에도 실패한다면 리스트의 인덱스 탐색을 시도하게 됩니다.

{% for %} 반복 구문에서 메소드 호출이 일어납니다. **question.choice_set.all**은 Python에서 **question.choice_set.all()** 코드로 해석되는데, 이때 반환된 **Choice** 객체의 반복자는 **{% for %}**에서 사용하기 적당합니다.

템플릿에 대한 더 많은 정보는 템플릿 지침서를 참고하세요.

템플릿에서 하드코딩된 URL 제거하기

polls/index.html 템플릿에 링크를 적으면, 이 링크는 다음과 같이 부분적으로 하드코딩된다는 것을 기억하세요.

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

이러한 강력하게 결합되고 하드코딩된 접근방식의 문제는 수 많은 템플릿을 가진 프로젝트들의 URL을 바꾸는 게 어려운 일이 된다는 점입니다. 그러나, **polls.urls** 모듈의 **path()** 함수에서 인수의 이름을 정의했으므로, **{% url %}** template 태그를 사용하여 url 설정에 정의된 특정한 URL 경로들의 의존성을 제거할 수 있습니다.

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

이것이 **polls.urls** 모듈에 서술된 URL 의 정의를 탐색하는 식으로 동작합니다. 다음과 같이 〈detail〉 이라는 이름의 URL 이 어떻게 정의되어 있는지 확인할 수 있습니다.

```
...
# the 'name' value as called by the {% url %} template tag
path('<int:question_id>/', views.detail, name='detail'),
...
```

만약 상세 뷰의 URL을 **polls/specifics/12/**로 바꾸고 싶다면, 템플릿에서 바꾸는 것이 아니라 **polls/urls.py**에서 바꿔야 합니다.

```
...
# added the word 'specifics'
path('specifics/<int:question_id>/', views.detail, name='detail'),
...
```

URL의 이름공간 정하기

튜토리얼의 프로젝트는 **polls**라는 앱 하나만 가지고 진행했습니다. 실제 Django 프로젝트는 앱이 몇개라도 올 수 있습니다. Django는 이 앱들의 URL을 어떻게 구별해 낼까요? 예를 들어, **polls** 앱은 **detail**이라는 뷰를 가지고 있고, 동일한 프로젝트에 블로그를 위한 앱이 있을 수도 있습니다. Django가 **{% url %}** 템플릿태그를 사용할 때, 어떤 앱의 뷰에서 URL을 생성할지 알 수 있을까요?

정답은 URLconf에 이름공간(namespace)을 추가하는 것입니다. **polls/urls.py** 파일에 **app_name**을 추가하여 어플리케이션의 이름공간을 설정할 수 있습니다.

polls/urls.py

```
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

이제, **polls/index.html** 템플릿의 기존 내용을

polls/templates/polls/index.html

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

아래와 같이 이름공간으로 나눠진 상세 뷰를 가리키도록 변경하세요.

polls/templates/polls/index.html

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

When you're comfortable with writing views, read part 4 of this tutorial to learn the basics about form processing and generic views.

‹ 첫 번째 장고 앱 작성하기, part 2

첫 번째 장고 앱 작성하기, part 4 ›

BACK TO TOP

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 2
- 다음: 첫 번째 장고 앱 작성하기, part 4

- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 3

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list
|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 4

This tutorial begins where Tutorial 3 left off. We're continuing the Web-poll application and will focus on form processing and cutting down our code.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

Write a minimal form

앞장의 투표 상세 템플릿(《polls/detail.html》)을 수정하여, 템플릿에 HTML **<form>** 요소를 포함시켜 봅시다.

polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id
}}">
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br>
{% endfor %}
<input type="submit" value="Vote">
</form>
```

간략하게 설명하면:

- 위의 템플릿은 각 질문 선택 항목에 대한 라디오 버튼을 표시합니다. 각 라디오 버튼의 **value**는 연관된 질문 선택 항목의 ID입니다. 각 라디오 버튼의 **name**은 **"choice"**입니다. 즉, 누군가가 라디오 버튼 중 하나를 선택하여 폼을 제출하면, POST 데이터 인 **choice=#**을 보낼 것입니다. 여기서 #은 선택한 항목의 ID입니다. 이것은 HTML 폼의 기본 개념입니다.
- We set the form's **action** to **{% url 'polls:vote' question.id %}**, and we set **method="post"**. Using **method="post"** (as opposed to **method="get"**) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use **method="post"**. This tip isn't specific to Django; it's good Web development practice in general.
- **forloop.counter** 는 **for** 태그가 반복을 한 횟수를 나타냅니다.
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a helpful system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the **{% csrf_token %}** template tag.

이제 제출된 데이터를 처리하고 그 데이터로 무언가를 수행하는 Django 뷰를 작성하겠습니다. 튜토리얼 3 에서 설문조사 어플리케이션을 위해 아래에 나와있는 코드를 포함하는 URLconf 를 만들었습니다:

polls/urls.py

```
path('<int:question_id>/vote/', views.vote, name='vote'),
```

또, 우리는 **vote()** 함수를 가상으로 만들었습니다. 실제로 구현을 해봅시다. **polls/views.py** 에 다음을 추가합시다:

polls/views.py

```
from django.http import HttpResponse, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

위 코드는 이 튜토리얼에서 아직 다루지 않은 몇 가지를 포함하고 있습니다:

- **request.POST** 는 키로 전송된 자료에 접근할 수 있도록 해주는 사전과 같은 객체입니다. 이 경우, **request.POST['choice']** 는 선택된 설문의 ID를 문자열로 반환합니다. **request.POST** 의 값은 항상 문자열들입니다.

  Django는 같은 방법으로 GET 자료에 접근하기 위해 **request.GET** 를 제공합니다 – 그러나 POST 요청을 통해서만 자료가 수정되게하기 위해서, 명시적으로 코드에 **request.POST** 를 사용하고 있습니다.

- 만약 POST 자료에 **choice** 가 없으면, **request.POST['choice']** 는 **KeyError** 가 일어납니다. 위의 코드는 **KeyError** 를 체크하고, choice가 주어지지 않은 경우에는 에러 메시지와 함께 설문조사 폼을 다시보여줍니다.

- 설문지의 수가 증가한 이후에, 코드는 일반 **HttpResponse** 가 아닌 **HttpResponseRedirect** 를 반환하고, **HttpResponseRedirect** 는 하나의 인수를 받습니다: 그 인수는 사용자가 재전송될 URL 입니다. (이 경우에 우리가 URL을 어떻게 구성하는지 다음 항목을 보세요).

  As the Python comment above points out, you should always return an **HttpResponseRedirect** after successfully dealing with POST data. This tip isn't specific to Django; it's good Web development practice in general.

- 우리는 이 예제에서 **HttpResponseRedirect** 생성자 안에서 **reverse()** 함수를 사용하고 있습니다. 이 함수는 뷰 함수에서 URL을 하드코딩하지 않도록 도와줍니다. 제어를 전달하기 원하는 뷰의 이름을, URL패턴의 변수부분을 조합해서 해당 뷰를 가리킵니다. 여기서 우리는 튜토리얼 3장에서 설정했던 URLconf를 사용하였으며, 이 **reverse()** 호출은 아래와 같은 문자열을 반환할 것입니다.

```
'/polls/3/results/'
```

여기서 `3` 은 `question.id` 값입니다. 이렇게 리디렉션된 URL은 최종 페이지를 표시하기 위해 `'results'` 뷰를 호출합니다.

튜토리얼 3장에서 언급했듯이, `request` 는 `HttpRequest` 개체입니다. `HttpRequest` 개체에 대해 더 알고 싶다면 request와 response 문서를 참고하세요.

어떤 이가 설문조사에 설문을 하고난 뒤에는, `vote()` 뷰는 설문조사 결과 페이지로 리다이렉트합니다. 그 뷰를 작성해봅시다:

polls/views.py

```
from django.shortcuts import get_object_or_404, render


def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

튜토리얼 3장의 `detail()` 뷰와 거의 동일합니다. 템플릿 이름만 다릅니다. 나중에 이 중복을 수정할 겁니다.

이제, `polls/results.html` 템플릿을 만듭니다.

polls/templates/polls/results.html

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize
}}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

이제, 웹 브라우저에서 `/polls/1/` 페이지로 가서, 투표를 해보세요. 당신이 투표를 할 때마다 값이 반영된 결과 페이지를 볼 수 있을 것입니다. 만약 당신이 설문지를 선택하지 않고 폼을 전송했다면, 오류 메시지를 보게 될 것입니다.

주석

우리의 `vote()` 뷰에는 작은 문제가 있습니다. 먼저 데이터베이스에서 `selected_choice` 객체를 가져온 다음, `votes` 의 새 값을 계산하고 나서, 데이터베이스에 다시 저장합니다. 만약 여러분의 웹사이트에 두 명의 사용자가 *정확하게 같은 시간* 에 투표를 할려고 시도할 경우, 잘못될 수 있습니다. `votes` 의 조회값이 42라고 할 경우, 두 명의 사용자에게 새로운 값인 43이 계산 되고, 저장됩니다. 그러나 44가 되야되겠죠.

이를 *경쟁 상태* 라 합니다. 이 문제를 해결할 수 있는 방법을 알아보려면 Avoiding race conditions using F() 를 참고하세요.

제너릭 뷰 사용하기: 적은 코드가 더 좋습니다.

The `detail()` (from Tutorial 3) and `results()` views are very short – and, as mentioned above, redundant. The `index()` view, which displays a list of polls, is similar.

이러한 뷰는 URL에서 전달 된 매개 변수에 따라 데이터베이스에서 데이터를 가져 오는 것과 템플릿을 로드하고 렌더링 된 템플릿을 리턴하는 기본 웹 개발의 일반적인 경우를 나타냅니다. Django는 이런 매우 일반적인 경우를 위해 《제너릭 뷰》 시스템이라는 지름길을 제공합니다.

제너릭 뷰는 일반적인 패턴을 추상화하여 앱을 작성하기 위해 Python 코드를 작성하지 않아도됩니다.

Let's convert our poll app to use the generic views system, so we can delete a bunch of our own code. We'll have to take a few steps to make the conversion. We will:

1. URLconf를 변환하십시오.
2. 불필요한 오래된보기 중 일부를 삭제하십시오.
3. Django의 제너릭 뷰를 기반으로 새로운 뷰를 도입하십시오.

자세한 내용은 계속 읽어 나가십시오.

왜 코드 셔플인가?

일반적으로 Django 앱을 작성할 때 일반 뷰가 문제에 적합한 지 여부를 평가할 것이며 코드를 중간에서 다시 리팩토링하지 않고 처음부터 사용하게됩니다. 그러나 이 튜토리얼은 의도적으로 현재까지 핵심 개념에 초점을 맞추기 위해 《어려운 방법》으로 뷰를 작성하는 데 중점을 두었습니다.

계산기를 사용하기 전에 기본 수학을 알아야합니다.

URLconf 수정

먼저, **polls/urls.py** URLconf를 열어 다음과 같이 변경하십시오:

polls/urls.py

```
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
    path('<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

두 번째와 세 번째 패턴의 경로 문자열에서 일치하는 패턴들의 이름이 **<question_id>** 에서 **<pk>** 로 변경되었습니다.

---

views 수정

다음으로 이전의 **index**, **detail**, **results**뷰를 제거하고 장고의 일반적인 뷰를 대신 사용하겠습니다. 그렇게하려면 **polls/views.py** 파일을 열고 다음과 같이 변경하십시오:

polls/views.py

```python
from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse
from django.views import generic

from .models import Choice, Question


class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]


class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'


class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'


def vote(request, question_id):
    ... # same as above, no changes needed.
```

ListView와 **DetailView**의 두 가지 제너릭 뷰를 사용하고 있습니다. 이 두보기는 각각 《개체 목록 표시》 및 《특정 개체 유형에 대한 세부 정보 페이지 표시》 개념을 추상화합니다.

- 각 제너릭 뷰는 어떤 모델이 적용될 것인지를 알아야합니다. 이것은 **model** 속성을 사용하여 제공됩니다.
- **DetailView** 제너릭 뷰는 URL에서 캡처된 기본 키 값이 **"pk"**라고 기대하기 때문에 **question_id**를 제너릭 뷰를 위해 **pk**로 변경합니다.

기본적으로 **DetailView** 제너릭 뷰는 **<app name>/<model name>_detail.html** 템플릿을 사용합니다. 우리의 경우에는 **"polls/question_detail.html"**템플릿을 사용할 것입니다. **template_name** 속성은 Django에게 자동 생성 된 기본 템플릿 이름 대신에 특정 템플릿 이름을 사용하도록 알려주기 위해 사용됩니다. **results**리스트 뷰에 대해서 **template_name**을 지정합니다 - 결과 뷰와 상세 뷰가 렌더링 될 때 서로 다른 모습을 갖도록합니다. 이들이 둘다 동일한 **DetailView**를 사용하고 있더라도 말이지요.

마찬가지로, **ListView** 제네릭 뷰는 **<app name>/<model name>_list.html** 템플릿을 기본으로 사용합니다; 이미 있는 **"polls/index.html"** 템플릿을 사용하기 위해 **ListView** 에 **template_name** 를 전달했습니다.

In previous parts of the tutorial, the templates have been provided with a context that contains the **question** and **latest_question_list** context variables. For **DetailView** the **question** variable is provided automatically – since we're using a Django model (**Question**), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is **question_list**. To override this we provide the **context_object_name** attribute, specifying that we want to use **latest_question_list** instead. As an alternative approach, you could change your templates to match the new default context variables – but it's a lot easier to tell Django to use the variable you want.

서버를 실행하고 제너릭 뷰를 기반으로한 새 설문조사 앱을 사용하십시오.

제너릭 뷰에 대한 자세한 내용은 제너릭 뷰 문서를 참조하십시오.

폼 및 제너릭 뷰가 마음에 들면, 이 설문조사 앱의 테스트에 대해 튜토리얼의 5장을 읽어 배워 보기 바랍니다.

‹ 첫 번째 장고 앱 작성하기, part 3

첫 번째 장고 앱 작성하기, part 5 ›

BACK TO TOP

Support Django!

TutorCruncher donated to the Django Software Foundation to support Django development. Donate today!

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 3
- 다음: 첫 번째 장고 앱 작성하기, part 5
- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 4

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차

Handy when looking for specific information.

---

django-users mailing list

|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel

Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker

Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity Statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program

Hosting by
rackspace.

Design by
threespot.
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 5

이 튜토리얼은 튜토리얼 4장에서 이어집니다. 우리는 웹 설문조사 애플리케이션을 구축했으며 이제 이를 위한 자동화된 테스트를 작성할 것입니다.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

자동화된 테스트 소개

자동화된 테스트란 무엇입니까?

Tests are routines that check the operation of your code.

테스트는 다양한 수준에서 작동합니다. 일부 테스트는 작은 세부 사항에 적용될 수 있습니다 (*특정 모델 메서드는 예상대로 값을 반환합니까?*) 또 다른 테스트는 소프트웨어의 전반적인 작동을 검사합니다 (*사이트에서 사용자 입력 시퀀스가 원하는 결과를 생성합니까?*). 이것은 이전 튜토리얼 2장에서 `shell`을 사용하여 메소드의 동작을 검사하거나 애플리케이션을 실행하고 어떻게 작동하는지 확인하기 위해 데이터를 입력해서 테스트했던 것과 다르지 않습니다.

자동화 된 테스트에서 다른 점은 테스트 작업이 시스템에서 수행된다는 것입니다. 한 번 테스트 세트를 작성한 이후에는 앱을 변경할 때 수동 테스트를 수행하지 않아도 원래 의도대로 코드가 작동하는지 확인할 수 있습니다.

테스트를 만들어야하는 이유

그래서 왜 테스트를 만들어야하고, 굳이 지금 해야할까요?

아마 당신은 Python/Django를 배우는 것 만으로도 충분하다고 느낄수 있고 또 다른것을 배우고 써보는것은 지나치거나 불필요해 보일수 있습니다. 어쨌든 간에 우리의 설문조사 어플리케이션은 지금 꽤 잘 돌아고 있습니다. 문제들을 해쳐 나가며 자동화된 테스트를 만드는게 이 어플리케이션을 작동하게 하거나 더 좋게 하지는 않습니다. 만약 투표 어플리케이션을 만드는게 Django 프로그래밍의 최종 단계라면, 자동화된 테스트를 어떻게 만드는지 알 필요는 없습니다. 하지만 아직 더 많은것을 하려 한다면 지금이 자동화된 테스트 작성을 배우기 딱 좋은 시간입니다.

테스트를 통해 시간을 절약 할 수 있습니다.

특정 시점까지는 〈제대로 작동하는지 확인〉 하는것이 테스트로서 충분할 것 입니다. 더 정교한 어플리케이션에서는 구성 요소간에 수십 개의 복잡한 상호 작용이있을 수 있습니다.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 〈seems to work〉 could mean running through your code's functionality with twenty different variations of your test data to make sure you haven't broken something - not a good use of your time.

이 수동 테스트 작업을 자동화된 테스트가 몇초만에 해낼수 있다면 귀한시간을 많이 아낄수 있겠죠?. 무언가가 잘못되어도 테스트를 통해 예기치 않은 동작을 일으키는 코드를 식별하는 데 도움이됩니다.

때로는 코드가 제대로 작동하고 있음을 알 때 테스트를 작성하는 것은 허드렛일로 보여서 당신의 생산적이고 창의적인 프로그래밍 작업에서 떠나 매력적이지도 흥분되지도 않는 테스트 작성이라는 일을 하는게 어려울수도 있습니다.

그러나 테스트를 작성하는 작업은 어플리케이션을 수동으로 테스트하거나 새로 발견된 문제의 원인을 확인하는 데 많은 시간을 투자하는 것보다 훨씬 더 효과적입니다.

테스트는 문제를 그저 식별하는 것이 아니라 예방합니다.

테스트를 그저 개발의 부정적 측면으로 생각하는 것은 실수입니다.

테스트가 없으면 어플리케이션의 목적 또는 의도 된 동작이 다소 불투명 할 수 있습니다. 심지어 자신의 코드 일 때도, 정확히 무엇을하고 있는지 알아 내려고 노력하게 됩니다.

테스트는 이 불투명함을 바꿉니다. 그들은 내부에서 코드를 밝혀 내고, 어떤 것이 잘못 될 때, *그것이 잘못되었다는 것을 깨닫지 못했다고 할지라도, 잘못된 부분에 빛을 집중시킵니다*.

테스트가 코드를 더 매력적으로 만듭니다.

You might have created a brilliant piece of software, but you will find that many other developers will refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says 《Code without tests is broken by design.》

테스트 작성을 시작해야하는 또다른 이유는 다른 개발자들이 당신의 소프트웨어를 사용하는것을 진지하게 고려하기 전에 테스트 코드를 보기를 원하기 때문입니다.

테스트는 팀이 함께 일하는것을 돕습니다

이전의 내용은 어플리케이션을 유지 관리하는 단일 개발자의 관점에서 작성되었습니다. 복잡한 애플리케이션은 팀별로 유지 관리됩니다. 테스트는 동료가 실수로 코드를 손상시키지 않는다는 것을 보증합니다 (그리고 당신이 동료의 코드를 모르는새에 망가트리는것도). 장고 프로그래머로서 생계를 꾸려 나가려면 테스트를 잘해야합니다!

기초 테스팅 전략

테스트 작성에 대한 많은 접근법이 있습니다.

Some programmers follow a discipline called 《test-driven development》; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development formalizes the problem in a Python test case.

더 흔하게는, 테스팅 입문자들은 코드를 작성하고 시간이 흐른 뒤에 테스트들이 필요하다고 판단할 것입니다. 아마도 몇몇의 테스트는 더 빨리 작성하는 것이 나을지도 모릅니다. 하지만 시작하기에 너무 늦어서는 안 됩니다.

어디서부터 테스트를 작성해야 할지 종잡을 수 없을 때가 있습니다. 당신이 수천 줄의 파이썬 코드를 작성해 놓았다면, 테스트할 것을 고르는 것이 쉽지 않을지도 모릅니다. 그럴 때는 다음에 새로운 기능을 넣거나 버그를 수정하는 등, 코드를 변경할 일이 있을 때, 당신의 첫 테스트를 작성하는 것이 유익할 것입니다.

그러니 지금 당장 해봅시다.

첫 번째 테스트 작성하기

버그 식별하기

다행스럽게도 **`polls`** 어플리케이션에는 우리가 즉시 해결할 수있는 약간의 버그가 있습니다.

Question.was_published_recently() 메소드는 **Question**이 어제 게시된 경우 **True**를 반환(올바른 동작)할 뿐만 아니라 **Question**의 **pub_date** 필드가 미래로 설정되어 있을 때도 그렇습니다(틀린 동작).

shell을 사용해 미래의 날짜로 메소드를 실행해 버그를 확인합니다.

```
$ python manage.py shell
```

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

미래는 〈최근(recent)〉이 아니기 때문에 이는 분명히 잘못된 것입니다.

버그를 노출하는 테스트 만들기

문제를 테스트하기 위해 **shell** 에서 방금 수행한 작업은 자동화된 테스트에서 수행할 수 있는 작업이므로 자동화된 테스트로 바꾸도록 합시다.

애플리케이션 테스트는 일반적으로 애플리케이션의 **tests.py** 파일에 있습니다. 테스트 시스템은 **test** 로 시작하는 파일에서 테스트를 자동으로 찾습니다.

polls 어플리케이션의 **tests.py** 파일에 다음을 입력하십시오:

polls/tests.py

```
import datetime

from django.test import TestCase
from django.utils import timezone

from .models import Question


class QuestionModelTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

우리는 미래의 **pub_date**를 가진 **Question** 인스턴스를 생성하는 메소드를 가진 **django.test.TestCase** 하위 클래스를 생성했습니다. 그런 다음 **was_published_recently()**의 출력이 False가 되는지 확인했습니다.

테스트 실행

터미널에서 테스트를 실행합니다.

```
$ python manage.py test polls
```

그러면 다음과 같은 것을 볼 수 있습니다.

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionModelTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

다른 오류가 발생하나요?

만약 위 오류 대신 **NameError**가 발생한다면, 당신은 Part 2의 한 단계를 놓쳤던 것일지도 모릅니다. 이 단계에서 우리는 **datetime**과 **timezone**의 import를 **polls/models.py**에 추가하였습니다. 해당 섹션에서 임포트를 복사한 후, 테스트를 다시 돌려보시길 바랍니다.

무슨 일이 일어 났습니까?

- **manage.py test polls**는 **polls** 애플리케이션에서 테스트를 찾습니다.
- **django.test.TestCase** 클래스의 서브 클래스를 찾았습니다.
- 테스트 목적으로 특별한 데이터베이스를 만들었습니다.
- 테스트 메소드 - 이름이 **test**로 시작하는 것들을 찾습니다.
- **test_was_published_recently_with_future_question**에서 **pub_date**필드가 30일 미래인 **Question** 인스턴스를 생성했습니다
- ... **assertIs()** 메소드를 사용하여, 우리가 **False**가 반환되기를 원함에도 불구하고 **was_published_recently()** 가 **True**를 반환한다는 것을 발견했습니다.

테스트는 어떤 테스트가 실패했는지와 실패가 발생한 행까지 알려줍니다.

버그 수정

우리는 이미 문제가 무엇인지 알고 있습니다: **Question.was_published_recently()**는 **pub_date**가 미래에 있다면 **False**를 반환해야 합니다. **models.py**에서 날짜가 과거에 있을 때에만 **True**를 반환하도록 메소드를 수정하십시오.

polls/models.py

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

이제 테스트를 다시 실행합니다.

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

버그를 확인한 후에 우리는 이를 드러내는 테스트를 작성 하였으며 코드에서 버그를 수정하고 테스트를 통과했습니다.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

보다 포괄적인 테스트

우리가 여기있는 동안, 우리는 **was_published_recently()**메소드를 고정하는것 이상을 할수 있습니다; 사실 하나의 버그를 고치면서 다른 새로운 버그를 만들어 낸다면 분명 곤란할것입니다.

메소드의 동작을보다 포괄적으로 테스트하기 위해 동일한 클래스에 두 가지 테스트 메소드를 추가하십시오:

polls/tests.py

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

이제 우리는 **Question.was_published_recently()**가 과거, 최근, 미래의 질문에 대해 올바른 값을 반환한다는걸 확인시켜주는 세가지 테스트를 가졌습니다.

Again, **polls** is a minimal application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

뷰 테스트

설문조사 어플리케이션은 상당히 대충대충 만들어져 있습니다. 이 어플리케이션은 **pub_date**필드가 미래에있는 질문 까지도 포함하여 게시합니다. 이것을 개선 해야합니다. 미래로 **pub_date**를 설정하는 것은 그 시기가 되면 질문이 게시되지만 그때까지는 보이지 않는 것을 의미 해야 합니다.

뷰에 대한 테스트

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was an example of test-driven development, but it doesn't really matter in which order we do the work.

첫 번째 테스트에서 코드의 내부 동작에 대해 자세히 설명했습니다. 이 테스트에서는 웹 브라우저를 통해 사용자가 경험하는대로 동작을 확인하려고합니다.

버그를 수정하기 전에 우리가 사용 할 수있는 도구를 살펴 보겠습니다.

장고 테스트 클라이언트

Django는 뷰 레벨에서 코드와 상호 작용하는 사용자를 시뮬레이트하기위해 테스트 클라이언트 클래스 **Client**를 제공합니다. 이 테스트 클라이언트를 **tests.py**또는 **shell**에서 사용할 수 있습니다.

우리는 또다시 **shell**에서 시작할 것인데, **tests.py**에서 필요하지 않았던 두 가지 일을 해야 합니다. 첫 번째는 **shell**에서 테스트 환경을 구성하는 것입니다.

```
$ python manage.py shell
```

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

*response.context*와 같은 *response*의 추가적인 속성을 사용할수 있게 하기위해서 **setup_test_environment()**를 사용하여 템플릿 렌더러를 설치합니다. 이 메소드는 테스트 데이터베이스를 셋업하지 *않습니다*. 그렇기 때문에 테스트는 현재 사용중인 데이터베이스 위에서 돌게되며 결과는 데이터베이스에 이미 만들어져있는 질문들에 따라서 조금씩 달라질 수 있습니다. 또한 **settings.py**의 **TIME_ZONE**이 올바르지 않으면 예기치 않은 결과가 발생할 수 있습니다. 초기에 어떻게 설정해놨는지 기억나지 않는다면 진행하기 전에 먼저 확인하십시오.

다음으로 우리는 테스트 클라이언트 클래스를 import 해야합니다. (나중에 **tests.py**에서는 **django.test.TestCase** 클래스에 같이 딸려오는 클라이언트를 사용할 것이므로 이것은 필요하지 않을 것입니다):

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

이런것들이 준비가 되었다면 이제 우리는 클라이언트에세 우리를 위해 일을 하라고 시킬수 있습니다.

```
>>> # get a response from '/'
>>> response = client.get('/')
Not Found: /
>>> # we should expect a 404 from that address; if you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
>>> # omitted the setup_test_environment() call described earlier.
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a href="/polls/1/">What&#x27;s up?</a></li>\n    \n
</ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>
```

뷰를 개선시키기

설문 조사 목록에는 아직 게시되지 않은 설문 조사 (즉, 장래에 **pub_date**가 있는 설문 조사)가 표시됩니다. 그것을 수정합시다.

튜토리얼 4장에서 **ListView** 클래스 기반 뷰를 소개했습니다.

polls/views.py

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]
```

우리는 **get_queryset()** 메소드를 수정하여 **timezone.now()**와 비교하여 날짜를 검사하도록 변경해야 합니다. 먼저 가져 오기를 추가해야 합니다.

polls/views.py

```
from django.utils import timezone
```

그리고 다음과 같이 **get_queryset** 메소드를 수정해야합니다:

polls/views.py

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter (pub_date__lte = timezone.now ())는 **timezone.now**보다 **pub_date**가 작거나 같은 **Question**을 포함하는 queryset을 반환합니다.

새로운 뷰 테스트

이제 **runserver**를 실행하면 브라우저에 사이트가 적재되고 과거와 미래 날짜의 **Questions**이 생성되고, 퍼블리시된 것들만 리스트에 나타나는 것을 확인으로써 예상대로 동작하는 것에 만족할 수 있을 것입니다. 앞으로 *어떤 변화를 일으키더라도 이러한 점에 영향을 끼치지 않도록* 하고 싶을 것이므로, 위의 **shell**에 기초를 둔 테스트를 작성합시다.

polls/tests.py에 다음을 추가하십시오:

polls/tests.py

```
from django.urls import reverse
```

새로운 테스트 클래스와 함께 질문들을 생성하는 함수를 만들 것입니다.

polls/tests.py

```
def create_question(question_text, days):
    """
    Create a question with the given `question_text` and published the
    given number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text, pub_date=time)


class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_past_question(self):
        """
        Questions with a pub_date in the past are displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list']
```

```
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future aren't displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past question 1.>']
        )
```

이 중 일부를 더 자세히 살펴 보겠습니다.

먼저, 질문 생성 함수인 **create_question**은 테스트 과정 중 설문을 생성하는 부분에서 반복 사용합니다.

test_no_questions는 질문을 생성하지는 않지만 《사용가능한 투표가 없습니다.》라는 메시지 및 **latest_question_list**가 비어 있음을 확인합니다. **django.test.TestCase** 클래스는 몇 가지 추가적인 선언 메소드를 제공합니다. 이 예제에서 우리는 **assertContains()**와 **assertQuerysetEqual()**을 사용합니다.

test_past_question에서 우리는 질문을 생성하고 그 질문이 리스트에 나타나는지 확인합니다.

test_future_question에서 우리는 미래의 **pub_date**로 질문을 만듭니다. 데이터베이스는 각 테스트 메소드마다 재설정되므로 첫 번째 질문은 더 이상 존재하지 않으므로 다시 인덱스에 질문이 없어야 합니다.

요컨데, 사이트에서 관리자 입력 및 사용자 경험에 대한 이야기를 하는 테스트를 만들었고, 모든 상태와 시스템 상태의 모든 새로운 변경 사항에 대해 예상하는 결과가 출력되는지 확인합니다.

DetailView 테스트하기

우리가 만든 것이 잘 작동합니다. 그러나 미래의 설문들은 *목록*에 나타나지는 않지만, 사용자가 URL을 알고 있거나, 추측하면 접근할 수 있습니다. 그래서 우리는 **DetailView**에 비슷한 제약 조건을 추가할 필요가 있습니다.

polls/views.py

```
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

We should then add some tests, to check that a **Question** whose **pub_date** is in the past can be displayed, and that one with a **pub_date** in the future is not:

polls/tests.py

```
class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=5)
        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
        """
        The detail view of a question with a pub_date in the past
        displays the question's text.
        """
        past_question = create_question(question_text='Past Question.', days=-5)
        url = reverse('polls:detail', args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

더 많은 테스트를위한 아이디어

우리는 비슷한 **get_queryset** 메소드를 **ResultsView**에 추가하고 그 뷰에 대한 새로운 테스트 클래스를 생성해야합니다. 그것은 우리가 방금 만든 것과 매우 유사합니다. 사실 계속 반복 작업을 할 것입니다.

테스트를 추가하면서 다른 방법으로 애플리케이션을 개선 할 수도 있습니다. 예를 들어, **선택 사항**이 없는 사이트에 **설문**을 게시 할 수 있다는 것은 바보같은 일입니다. 그래서 우리의 뷰는 이를 확인하고 그러한 질문을 배제 할 것입니다. 우리의 테스트는 **선택사항**이 없는 **설문**을 생성 한 다음, 실제로 게시되지 않는지 테스트하고, **선택사항**이 있는 **설문**을 작성하고 게시 여부를 테스트합니다.

아마도 일반 사용자가 아닌 로그인 한 관리자는 게시되지 않은 **설문**을 볼 수 있어야합니다. 다시 말하면: 소프트웨어를 추가하기 위해 필요한 것은 무엇이든 테스트를 수반해야합니다, 먼저 테스트를 작성한 다음 코드가 테스트를 통과하게 만들 것인지, 아니면 먼저 코드에서 로직을 처리 한 다음 이를 증명할 테스트를 작성하십시오.

어느 순간엔가 너무 많은 테스트 코드들을 보고 관리하기 힘들도록 너무 비대해 지는것은 아닌가 생각할수도 있습니다.

테스트 할 때는, 많이 할 수록 좋습니다.

우리의 테스트가 통제 불능으로 성장하고있는 것처럼 보일 수 있습니다. 이 속도라면 곧 우리의 어플리케이션에서 보다 우리의 테

스트의 코드가 더 많아질 것이고, 나머지 코드의 우아한 간결함과 비교했을 때, 반복하는 것은 미학적입니다.

사실 비대해지는것은 중요하지 않습니다. 테스트 코드들이 늘어나게 하십시오. 대부분의 경우 테스트를 한 번 작성한 다음 신경을 끄게 됩니다. 그래도 이 테스트 코드의 유용한 기능들은 프로그램을 개발하는 동안 계속 해서 작동할것입니다.

때로는 테스트를 업데이트해야합니다. 우리가 **선택지**를 가진 **설문들**만 출력되도록 뷰를 수정한다고 가정 해 보겠습니다. 이 경우 기존 테스트 중 상당수가 실패 할 것입니다. *테스트 결과를 최신으로 유지하기 위해 어떤 테스트를 수정해야하는지 정확하게 알려주므로* 테스트가 스스로를 돌보는 데 도움이됩니다.

최악의 경우 개발을 계속할 때 중복되는 테스트가 있을 수 있습니다. 그것은 문제가 아닙니다. 테스팅에서 반복하는 것은 *좋은* 일입니다.

테스트들이 현명하게 배열되어있는 한 관리가 어려워지지 않을 것입니다. 경험에 근거한 좋은 방법 중에는 다음과 같은 내용이 있습니다.

- 각 모델이나 뷰에 대한 별도의 `TestClass`
- 테스트하려는 각 조건 집합에 대해 분리된 테스트 방법
- 기능를 설명하는 테스트 메소드 이름

추가 테스팅

이 튜토리얼에서는 테스트의 기본 사항에 대해서만 소개합니다. 여러분은 더 많은 것을 할 수도 있고, 또 사용할 수 있는 똑똑한 도구들이 많이 있습니다.

예를 들어, 이 전에 수행 한 테스트에서는 모델의 내부 로직과 뷰에서 정보를 게시하는 방법을 다루었지만 Selenium 같은 《브라우저 내》 프레임 워크를 사용하여 HTML이 브라우저에서 실제로 렌더링되는 방식을 테스트 할 수 있습니다. 이러한 도구를 사용하면 장고 코드의 동작뿐만 아니라 JavaScript도 확인할 수 있습니다. 테스트가 브라우저를 시작하고 인간이 그것을 다루는 것처럼 사이트와 상호 작용 것은 매우 중요합니다! Django에는 `LiveServerTestCase`가 포함되어있어 Selenium과 같은 도구와 쉽게 통합할 수 있게 해줍니다.

복잡한 어플리케이션을 사용하는 경우 연속적으로 통합하기 위해 모든 커밋마다 자동으로 테스트를 실행하여 품질 제어가 적어도 부분적으로 자동화되도록 할 수 있습니다.

어플리케이션에서 테스트되지 않은 부분을 탐지하는 좋은 방법은 코드 커버리지를 확인하는 것입니다. 이것은 또한 깨지기 쉬운 코드나 심지어는 죽은 코드를 식별하는 데 도움이됩니다. 코드를 테스트 할 수 없다는 것은 대개 코드가 리팩터링해야하거나 제거해야 함을 의미합니다. 커버리지는 죽은 코드를 확인하는 데 도움이됩니다. 자세한 내용은 Integration with coverage.py 를 참조하십시오.

장고 테스트는 테스트에 대한 포괄적인 정보를 제공합니다.

다음 내용은?

테스트에 대한 자세한 내용은 장고 테스트를 참조하십시오.

Django 뷰를 테스트하는 것이 익숙해졌으면, 정적 파일 관리에 대해 배울 수 있는 이 튜토리얼의 6장을 보세요.

‹ 첫 번째 장고 앱 작성하기, part 4

첫 번째 장고 앱 작성하기, part 6 ›

BACK TO TOP

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 4
- 다음: 첫 번째 장고 앱 작성하기, part 6
- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 5

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list
|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 6

이 튜토리얼은 튜토리얼 5장에 이어서 시작합니다. 우리는 테스트 된 웹 설문조사 애플리케이션을 구축했으며, 이제 스타일 시트와 이미지를 추가 할 것입니다.

서버에서 생성 된 HTML을 제외하고, 웹 어플리케이션은 일반적으로 전체 웹 페이지를 렌더링하는 데 필요한 추가 파일 – 예:이미지, JavaScript 또는 CSS – 을 제공해야합니다. Django에서는 이러한 파일을 《정적 파일》 이라고 부릅니다.

For small projects, this isn't a big deal, because you can keep the static files somewhere your web server can find it. However, in bigger projects – especially those comprised of multiple apps – dealing with the multiple sets of static files provided by each application starts to get tricky.

이것이 **`django.contrib.staticfiles`**의 목적입니다: 이것은 각 응용 프로그램(및 여러분이 지정한 다른 위치)의 정적 파일들을 프로덕션 환경에서 쉽게 제공 할 수있는 단일 위치로 수집합니다.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

*앱*의 모양과 느낌을 원하는 대로 바꿔보세요.

먼저, **`polls`** 디렉토리에 **`static`** 디렉토리를 만듭니다. **`polls/templates/`** 안의 템플릿을 찾는 것과 비슷하게 정적 파일을 찾습니다.

Django의 **`STATICFILES_FINDERS`** 설정은 다양한 소스에서 정적 파일을 찾는 방법을 알고 있는 파인더 목록을 가지고 있습니다. 기본값 중 하나는 **`AppDirectoriesFinder`** 인데, **`INSTALLED_APPS`** 에서 《정적》 하위 디렉토리를 찾습니다, 방금 생성 한 **`polls`** 의 경우입니다. 관리 사이트는 정적 파일에 대해 동일한 디렉토리 구조를 사용합니다.

Within the **`static`** directory you have just created, create another directory called **`polls`** and within that create a file called **`style.css`**. In other words, your stylesheet should be at **`polls/static/polls/style.css`**. Because of how the **`AppDirectoriesFinder`** staticfile finder works, you can refer to this static file in Django as **`polls/style.css`**, similar to how you reference the path for templates.

정적 파일 네임스페이싱

Just like templates, we *might* be able to get away with putting our static files directly in **`polls/static`** (rather than creating another **`polls`** subdirectory), but it would actually be a bad idea. Django will choose the first static file it finds whose name matches, and if you had a static file with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best way to ensure this is by *namespacing* them. That is, by putting those static files inside *another* directory named for the application itself.

그 스타일 시트(**`polls/static/polls/style.css`**)에 다음 코드를 넣으세요.

```
polls/static/polls/style.css
```

```
li a {
    color: green;
}
```

다음으로, **polls/templates/polls/index.html**의 맨 위에 다음을 추가하세요.

polls/templates/polls/index.html

```
{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}">
```

{% static %} 템플릿 태그는 정적 파일의 절대 URL을 생성합니다.

개발에 필요한 것은 이것이 전부입니다.

서버를 시작합니다(이미 실행 중이라면 재시작합니다).

```
$ python manage.py runserver
```

http://localhost:8000/polls/를 새로고침하면 질문의 링크가 녹색(Django 스타일!)로 표시되는 것을 볼 수 있으며, 이는 스타일시트가 올바로 적재된 것을 의미합니다.

배경 이미지 추가하기

다음으로, 이미지 용 하위 디렉토리를 만듭니다. **polls/static/polls/** 디렉토리에 **images** 서브 디렉토리를 만듭니다. 이 디렉토리 안에 **background.gif**라는 이미지를 넣으십시오. 즉, 이미지를 **polls/static/polls/images/background.gif**에 넣으십시오.

그런 다음, 스타일 시트(**polls/static/polls/style.css**)에 다음을 추가하세요.

polls/static/polls/style.css

```
body {
    background: white url("images/background.gif") no-repeat;
}
```

http://localhost:8000/polls/을 새로고침하면 화면의 왼쪽 상단에 배경이 나타날 것입니다.

> **경고**
>
> The **{% static %}** template tag is not available for use in static files which aren't generated by Django, like your stylesheet. You should always use **relative paths** to link your static files between each other, because then you can change **STATIC_URL** (used by the **static** template tag to generate its URLs) without having to modify a bunch of paths in your static files as well.

지금까지의 내용은 **기본**입니다. 프레임워크에 포함된 설정 및 다른 것들에 대한 자세한 내용은 정적 파일 howto와 정적 파일 레퍼런스를 참조하십시오. 정적 파일 배포는 실제 서버에서 정적 파일을 사용하는 방법을 설명합니다.

정적 파일에 익숙해졌으면, Django의 자동 생성되는 관리자 사이트를 커스터마이징하는 법에 대해 배울 수 있는 이 튜토리얼의 7장을 보세요.

Support Django!

Qukio donated to the Django Software Foundation to support Django development.
Donate today!

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 5
- 다음: 첫 번째 장고 앱 작성하기, part 7
- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 6

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list

|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity Statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Support Us

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**

첫 번째 장고 앱 작성하기, part 7

이 튜토리얼은 튜토리얼 6장에 이어서 시작합니다. 우리는 웹 설문조사 어플리케이션을 계속 사용하고 있으며, 튜토리얼 2장에서 봤었던 자동으로 생성된 관리자 사이트를 커스터마이징하는 데 초점을 맞출 것입니다.

도움을 받을 수 있는 방법

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

관리자 폼 커스터마이징

Question 모델을 **admin.site.register(Question)**에 등록함으로써, Django는 디폴트 폼 표현을 구성 할 수 있었습니다. 관리 폼이 보이고 작동하는 방법을 커스터마이징하려는 경우가 있습니다. 객체를 등록 할 때 Django에 원하는 옵션을 알려주면 커스터마이징 할 수 있습니다.

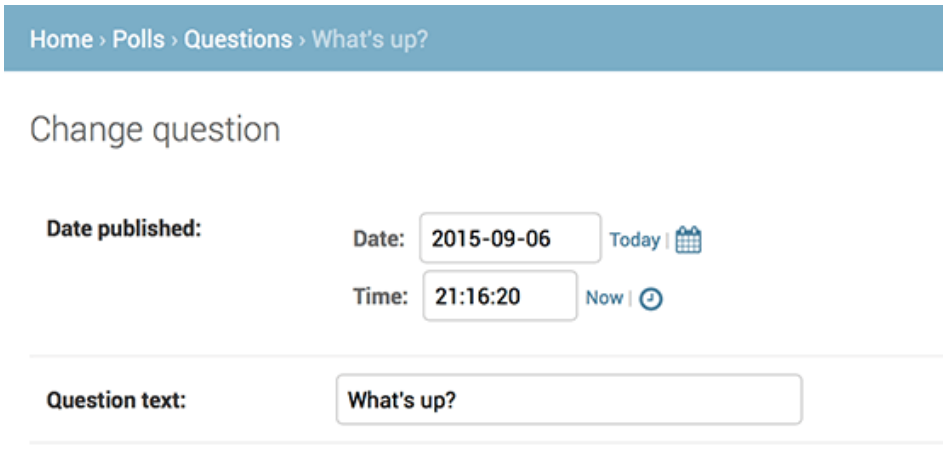
수정 폼의 필드를 재정렬하여 이것이 작동하는 법을 보겠습니다. **admin.site.register(Question)** 줄을 다음과 같이 바꾸세요:

polls/admin.py

```
from django.contrib import admin

from .models import Question


class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```

모델의 관리자 옵션을 변경해야 할 때마다 이 패턴 – 모델 어드민 클래스를 만든 다음, **admin.site.register()**에 두 번째 인수로 전달합니다 – 을 따라하면 됩니다.

여기서 특별한 변경 사항은 《발행일》이 《설문》 필드 앞에 오게 만듭니다.



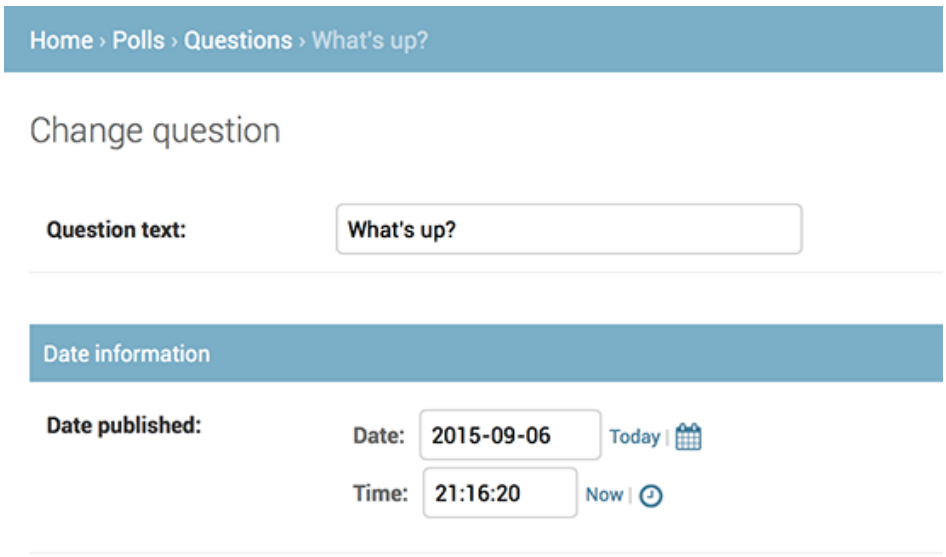
단지 2개의 필드만으로는 인상적이지는 않지만, 수십 개의 필드가 있는 관리 폼의 경우에는 직관적인 순서을 선택하는 것이 사용 편리성의 중요한 부분입니다.

수십 개의 필드가 있는 폼에 관해서는 폼을 fieldset으로 분할하는 것이 좋습니다.

polls/admin.py

```
from django.contrib import admin

from .models import Question


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```

fieldsets의 각 튜플의 첫 번째 요소는 fieldset의 제목입니다. 우리의 폼이 다음과 같이 변했네요.



관련된 객체 추가

OK, 우리는 Question 관리자 페이지를 가지고 있습니다. 그러나, **Question**은 여러 개의 **Choice**들을 가지고 있음에도, admin 페이지는 선택 사항을 표시하지 않습니다.

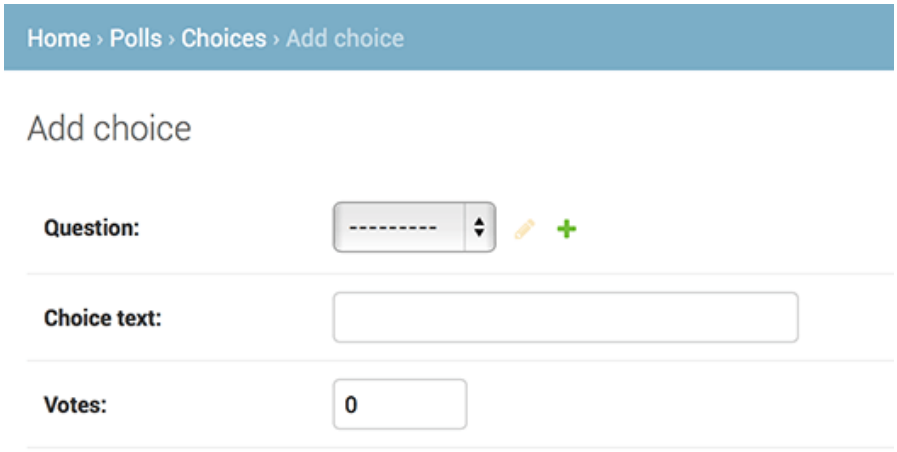
아직까진 말이죠.

There are two ways to solve this problem. The first is to register **Choice** with the admin just as we did with **Question**:

polls/admin.py

```
from django.contrib import admin

from .models import Choice, Question
# ...
admin.site.register(Choice)
```

이제 《Choices》는 Django 관리자가 사용할 수 있는 옵션입니다. 《Add choice》 폼은 다음과 같습니다.



이 양식에서 《Question》 필드는 데이터베이스의 모든 질문을 포함하는 select box입니다. Django는 **ForeignKey**가 admin에서 **<select>**로 표현되어야 함을 알고 있습니다. 우리의 경우, 지금은 단 하나의 질문만이 존재합니다.

또한 《Question》 옆의 《Add Another》 링크를 주목하세요. **ForeignKey** 관계를 가진 모든 객체는 저 링크가 붙습니다. 《Add Another》를 클릭하면 《Add question》 폼이 있는 팝업 창이 나타납니다. 해당 창에 질문을 추가하고 《Save》를 클릭하면 장고는 질문을 데이터베이스에 저장하고, 동적으로 이를 선택된 항목으로 당신이 보고있는 《Add choice》 폼에 추가합니다.

그러나 실제로 이것은 **Choice** 객체를 시스템에 추가하는 비효율적인 방법입니다. **Question** 객체를 생성할 때 여러 개의 Choices를 직접 추가할 수 있다면 더 좋을 것입니다. 그것을 만들어 봅시다.

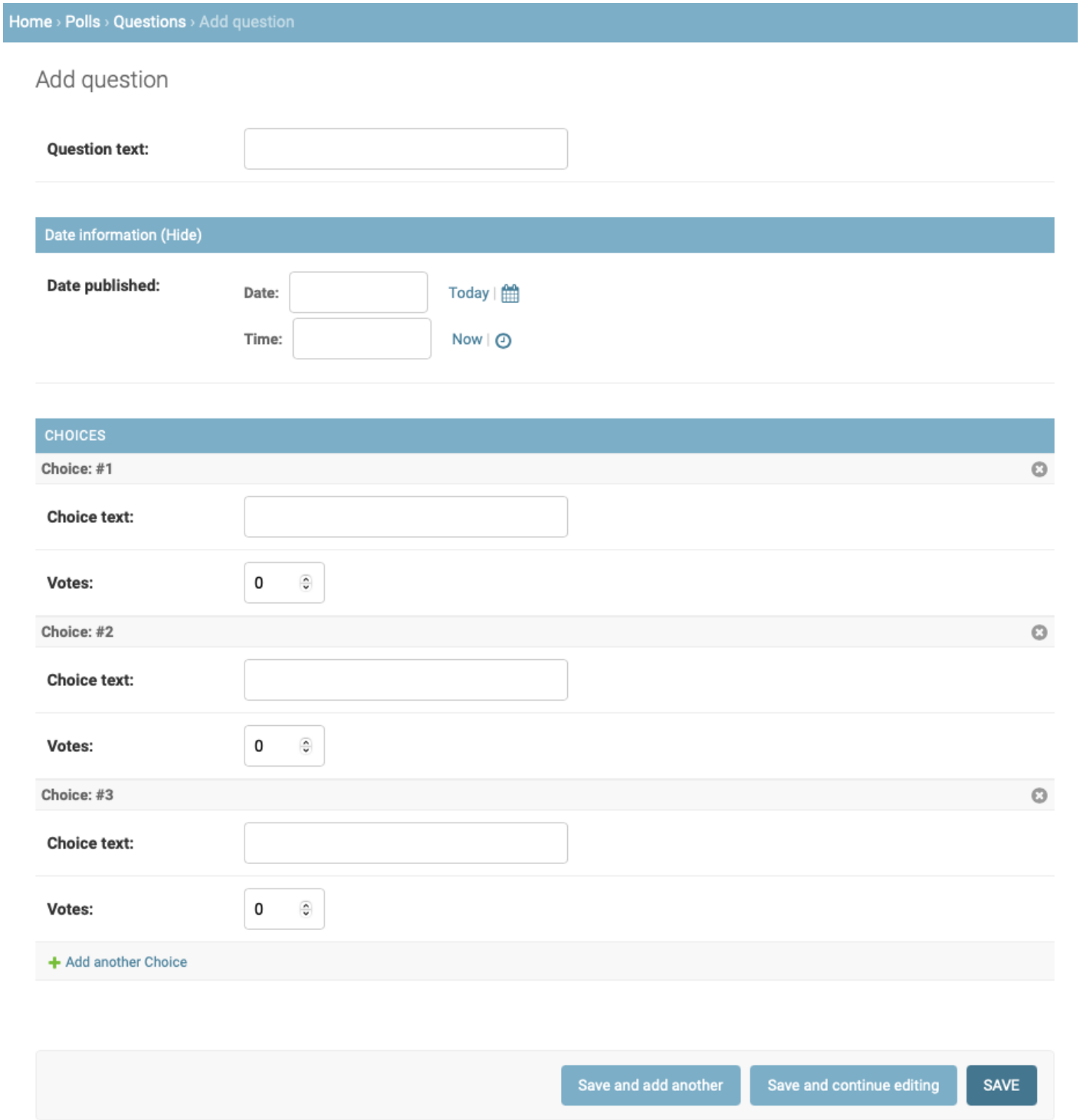
Choice 모델에 대한 **register()** 호출을 제거하십시오. 그런 다음 **Question** 등록 코드를 다음과 같이 편집하십시오:

polls/admin.py

```
from django.contrib import admin

from .models import Choice, Question


class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)
```

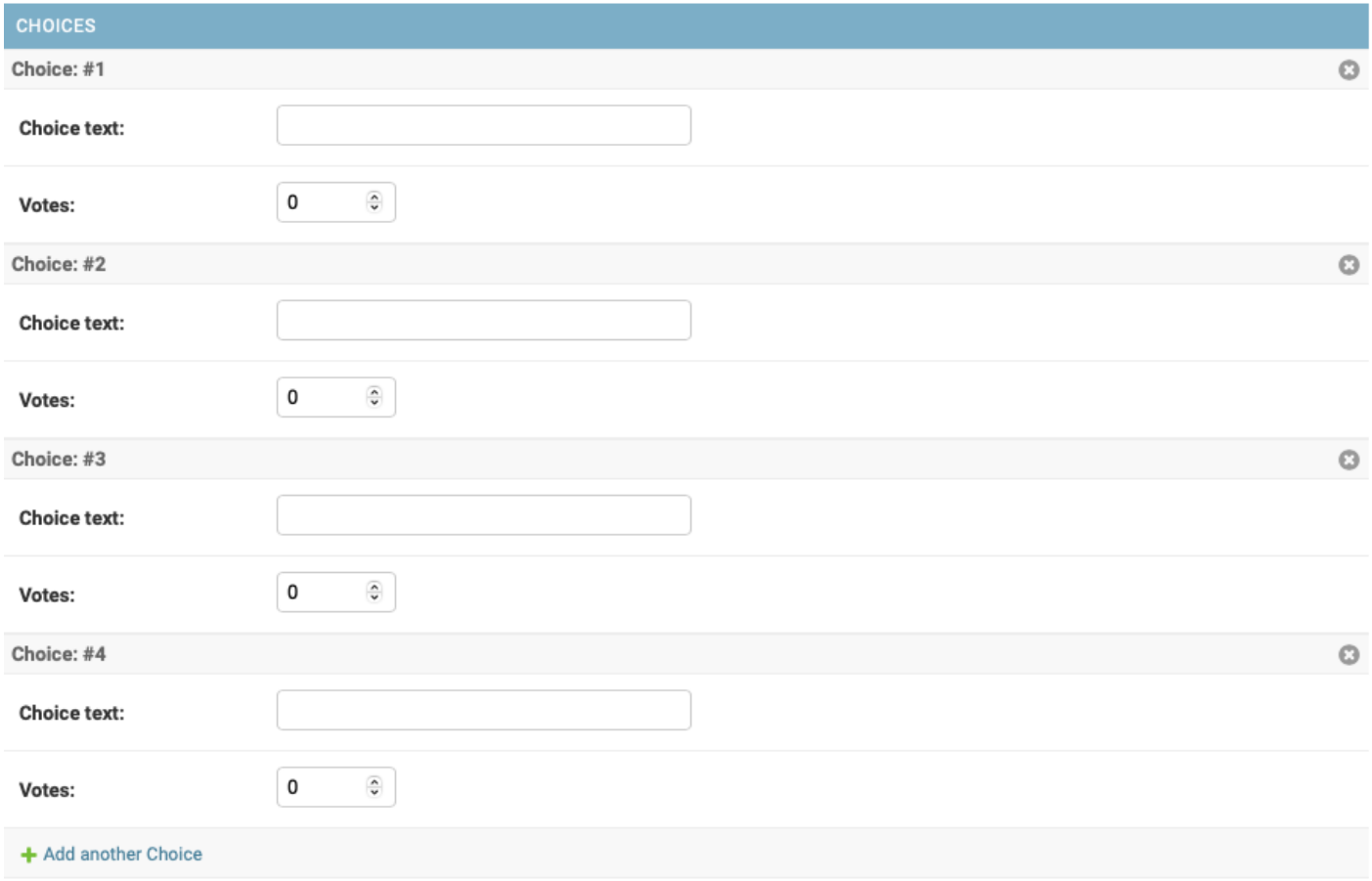
위 소소는 Django에게 《**`Choice`** 객체는 **`Question`** 관리자 페이지에서 편집된다. 기본으로 3가지 선택 항목을 제공함.》 이라고 알려줍니다.

모양을 보려면 《Add question》 페이지를 로드하십시오.



이 화면은 다음과 같이 작동합니다. 관련된 선택 사항을 위한 슬롯이 세 개 있으며– **`extra`**로 지정됨 – 이미 생성된 객체의 《Change》 페이지의 경우에도 빈 세 개의 슬롯이 생깁니다.

At the end of the three current slots you will find an 《Add another Choice》 link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. This image shows an added slot:

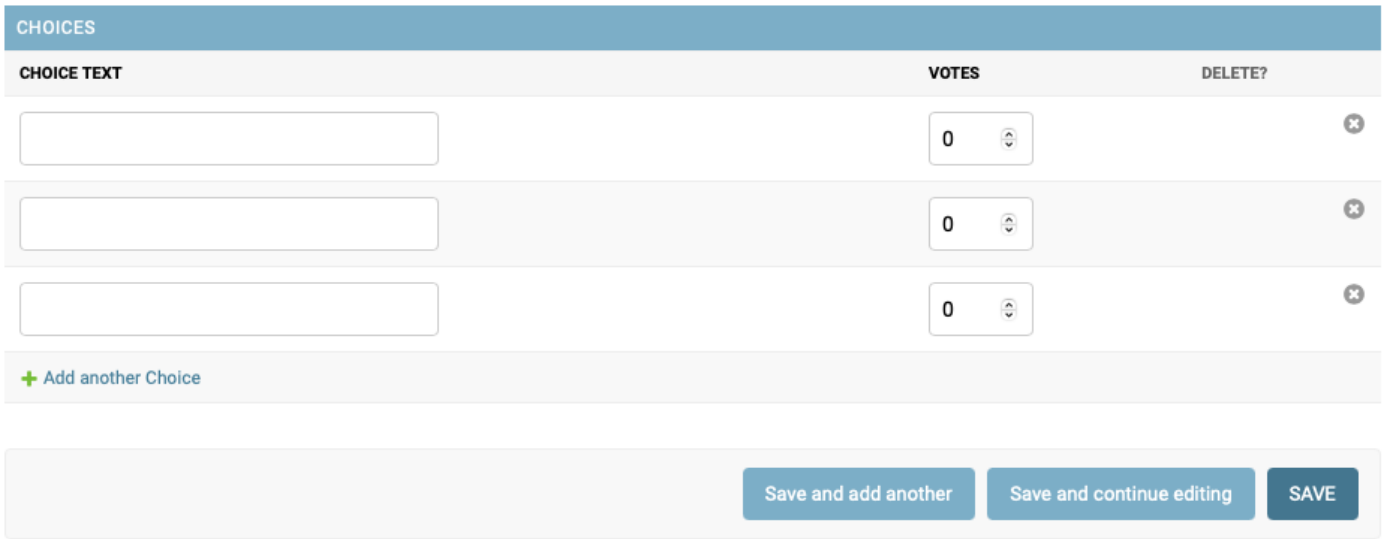


One small problem, though. It takes a lot of screen space to display all the fields for entering related **`Choice`** objects. For that reason, Django offers a tabular way of displaying inline related objects. To use it, change the **`ChoiceInline`** declaration to read:

polls/admin.py

```
class ChoiceInline(admin.TabularInline):
    #...
```

`StackedInline` 대신에 **`TabularInline`**을 사용하면, 관련된 객체는 좀 더 조밀하고 테이블 기반 형식으로 표시됩니다:

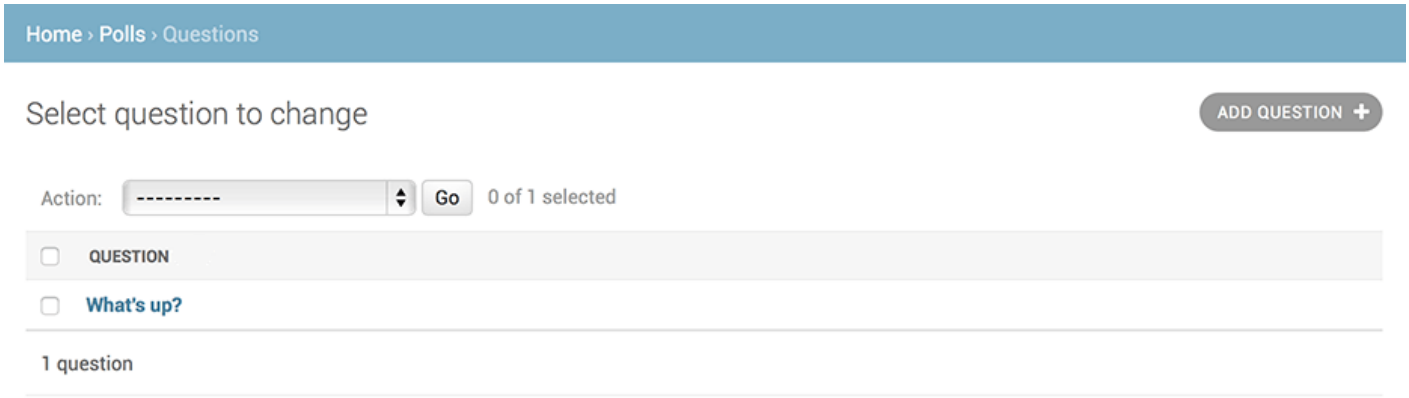


참고로 《Delete?》 열은 《Add Another Choice》 버튼으로 추가된 행과 이미 저장된 행을 삭제하는데 사용합니다.

관리자 변경 목록(change list) 커스터마이징

이제 질문 관리 페이지가 괜찮아 보이므로, 시스템의 모든 질문을 표시하는 《변경 목록》 페이지를 약간 조정하십시오.

다음은 이 시점의 모습입니다.



기본적으로 Django는 각 객체의 **`str()`**을 표시합니다. 그러나 개별 필드를 표시 할 수 있는 경우 가끔 도움이 될 수 있습니다. 이렇게 하려면 **`list_display`** admin 옵션을 사용합니다. 이 옵션은 객체의 변경 목록 페이지에서 열로 표시 할 필드 이름들의 튜플입니다.

polls/admin.py

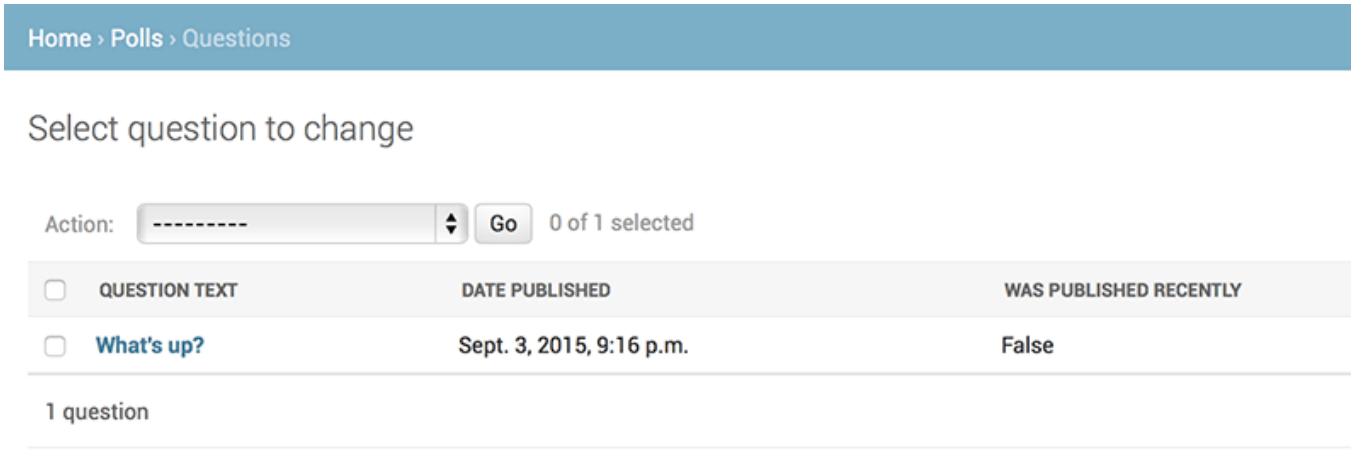
```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date')
```

For good measure, let's also include the **`was_published_recently()`** method from Tutorial 2:

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

이제 질문 변경 목록 페이지는 다음과 같습니다.



`was_published_recently` 헤더의 경우를 제외하고 그 값으로 정렬하기 위해 열 머리글을 클릭 할 수 있습니다. 왜냐하면 임의의 메서드의 출력에 의한 정렬은 지원되지 않기 때문입니다. 또한 **`was_published_recently`**에 대한 열 머리글은 기본적으로 메서드 이름 (밑줄을 공백으로 대체)이며 각 줄에는 출력의 문자열 표현이 포함되어 있습니다.

다음과 같이 해당 메소드 (**`polls/models.py`**)에 몇 가지 속성을 부여하여 향상시킬 수 있습니다:

polls/models.py

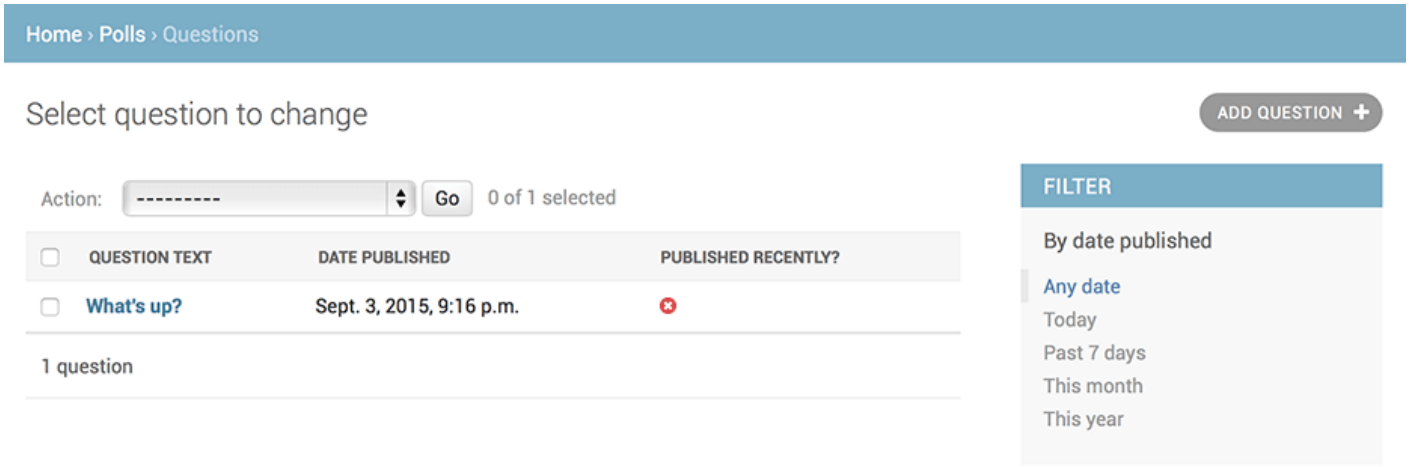
```
class Question(models.Model):
    # ...
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

이러한 메소드 속성들에 대한 더 자세한 정보는 **list_display**를 참조하십시오.

polls/admin.py 파일을 다시 편집하고 *Question* 변경 목록 페이지에 개선점을 추가하십시오: **list_filter**를 사용하는 필터. **QuestionAdmin**에 다음 줄을 추가하십시오:

```
list_filter = ['pub_date']
```

pub_date 필드에 의해 사람들이 변경 목록을 필터링 할 수 있게 해주는 《Filter》 사이드 바가 추가되었습니다:



표시되는 필터의 유형은 필터링중인 필드의 유형에 따라 다릅니다. **pub_date**는 **DateTimeField**이므로, Django는 《Any date》, 《Today》, 《Past 7 days》, 《This month》, 《This year》 등의 적절한 필터 옵션을 제공합니다.

잘 만들어지고 있습니다. 이제 검색 기능을 추가해 보겠습니다:

```
search_fields = ['question_text']
```

그러면 변경 목록 맨 위에 검색 창이 추가됩니다. 누군가가 검색어를 입력하면, 장고는 **question_text** 필드를 검색합니다. 당신이 원하는 만큼의 필드를 사용할 수 있습니다 – 그것은 내부적으로 **LIKE** 쿼리를 사용하기 때문에 검색 필드의 수를 적당한 수로 제한하면 데이터베이스가 검색을 더 쉽게 할 수 있습니다.

이제 변경 목록이 자동 페이징 기능을 제공한다는 점을 기억하십시오. 기본값은 페이지 당 100 개의 항목을 표시하는 것입니다. **변경 목록 페이지내이션**, **검색 상자**, **필터**, **날짜-계층구조**, 그리고 **컬럼-헤더-정렬** 모두 함께 작동합니다.

관리자 룩앤필 커스터마이징

명백히, 모든 관리자 페이지 상단에 《Django administration》이 있다는 것은 우스꽝스럽습니다. 이건 그저 자리를 채워넣기 위한 문자열입니다.

You can change it, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system.

프로젝트의 템플릿 커스터마이징

프로젝트 디렉토리 (**manage.py**를 포함하고있는)에 **templates** 디렉토리를 만듭니다. 템플릿은 장고가 액세스 할 수있는 파일 시스템 어디에서나 사용할 수 있습니다. (Django는 서버가 실행되는 사용자로 실행됩니다.) 그러나 프로젝트 내에 템플릿을 유지하는 것은 따라야 할 좋은 규칙입니다.

설정 파일 (**mysite/settings.py**를 기억하세요)을 열고 **DIRS** 옵션을 **TEMPLATES** 설정에 추가하십시오:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

DIRS는 Django 템플릿을 로드 할 때 검사 할 파일 시스템 디렉토리 목록입니다. 바로 검색 경로입니다.

템플릿 구성

정적 파일과 마찬가지로, 하나의 커다란 템플릿 디렉토리에 모든 템플릿을 함께 넣을 수 있습니다. 그렇게 해도 완벽하게 잘 작동할 것입니다. 그러나, 특정 애플리케이션에 속한 템플릿은 프로젝트(**templates**) 가 아닌 해당 애플리케이션의 템플릿 디렉토리(예: **polls/templates**)에 있어야 합니다. 우리는 왜 우리가 이렇게 해야하는 지에 대한 더 자세한 내용을 reusable apps tutorial 에서 논의할 것입니다.

이제 **templates** 디렉토리에 **admin**이라는 디렉토리를 만들고 장고 소스 코드 디렉토리 (**django/contrib/admin/templates**) 에 기본 관리자 템플릿 디렉토리 안에 있는 *admin/base_site.html*` 템플릿을 방금 만든 디렉토리로 복사합니다.

Django 소스 파일은 어디에 있습니까?

Django 소스 파일이 시스템에있는 위치를 찾는 데 어려움이있는 경우 다음 명령을 실행하십시오.

```
$ python -c "import django; print(django.__path__)"
```

Then, edit the file and replace **{{ site_header|default:_('Django administration') }}** (including the curly braces) with your own site's name as you see fit. You should end up with a section of code like:

```
{% block branding %}
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
{% endblock %}
```

이 방법을 사용하여 템플릿을 재정의하는 방법을 학습합니다. 실제 프로젝트에서는 아마 `django.contrib.admin.AdminSite.site_header` 속성을 사용하여 이 개별 커스터마이징을 보다 쉽게 만들 수 있습니다.

이 템플릿 파일에는 `{% block branding %}`및 `{{title}}`과 같은 텍스트가 많이 포함되어 있습니다. `{%`와 `{{`태그들은 장고의 템플릿 언어의 일부입니다. Django가 `admin/base_site.html`을 렌더링 할 때, 이 템플릿 언어는 튜토리얼 3장에서 보았 듯이 최종 HTML 페이지를 생성하기 위해 평가 될 것입니다.

Note that any of Django's default admin templates can be overridden. To override a template, do the same thing you did with `base_site.html` – copy it from the default directory into your custom directory, and make changes.

어플리케이션의 템플릿 사용자 정의

그러나 `DIRS`가 기본설정으로 비어 있다면, 장고는 기본 관리자 템플릿을 어떻게 찾을까요? 그 해답은 `APP_DIRS` 설정이 `True`로 설정되어 있기 때문에 Django는 각 어플리케이션 패키지 내에서 `templates/` 서브 디렉토리를 자동으로 찾아서 대체하게 됩니다. (`django.contrib.admin`이 어플리케이션 임을 잊지 마십시오.)

투표 어플리케이션은 복잡하지 않으며 사용자 정의 admin 템플릿이 필요하지 않습니다. 그러나 Django의 표준 admin 템플릿을 좀 더 정교하게 필요에 맞게 수정 할 경우 *프로젝트* 템플릿 대신 *어플리케이션*의 템플릿을 수정하는 것이 더 현명합니다. 그렇게 하면 다른 새 프로젝트에 투표 애플리케이션을 포함시킬 수 있고, 필요할때는 커스텀 템플릿을 찾을수 있습니다.

Django가 템플릿을 찾는 방법에 대한 자세한 정보는 템플릿 로딩 문서를 보십시오.

---

admin 인덱스 페이지 수정하기

제목과 같이, Django admin 인덱스 페이지의 모양과 느낌을 수정하고 싶을 수도 있습니다.

기본적으로 admin 어플리케이션과 함께 등록된 `INSTALLED_APPS`의 모든 어플리케이션을 사전순으로 표시합니다. 어쩌면 레이아웃을 크게 변경하고자 할 수 있습니다. 설사 그렇게 하더라도 인덱스는 admin의 가장 중요한 페이지이고, 사용하기 쉬워야 합니다.

커스터마이징 할 템플릿은 `admin/index.html`입니다. (이전 섹션의 `admin/base_site.html`와 같은 작업을 합니다 - 기본 디렉토리에서 커스텀 템플릿 디렉토리로 복사하십시오). 파일을 편집하면 `app_list`라는 템플릿 변수를 사용하는 것을 볼 수 있습니다. 이 변수는 설치된 모든 장고 앱을 포함합니다. 이를 사용하는 대신 최선의 방법이라고 생각한다면 개체 별 admin 페이지에 대한 링크를 하드 코딩 할 수 있습니다.

다음 내용은?

초보 자습서는 여기서 끝납니다. 아울러 앞으로 무엇을 해야할 지 문서를 좀 더 읽어볼 수 있습니다.

파이썬 패키징에 익숙하고 설문 조사를 《재사용 가능한 앱》으로 바꾸는 방법을 배우고 싶다면 심화 튜토리얼:재사용 가능한 애플리케이션을 만드는 법을 읽어보십시오.

‹ 첫 번째 장고 앱 작성하기, part 6

심화 튜토리얼: 재사용 가능한 앱을 만드는 법 ›

BACK TO TOP

목차

Browse

- 이전: 첫 번째 장고 앱 작성하기, part 6
- 다음: 심화 튜토리얼: 재사용 가능한 앱을 만드는 법
- 목차
- 전체 색인
- Python 모듈 목록

현재 위치:

- Django 3.1 문서
  - 시작하기
    - 첫 번째 장고 앱 작성하기, part 7

도움말

FAQ
공통적인 질문에 대한 답을 FAQ에서 찾아보세요.

색인, 모듈 색인, or 목차
Handy when looking for specific information.

django-users mailing list
|django-users 메일링 리스트 저장소나 공개된 질문에서 정보를 찾으세요

#django IRC channel
Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.

Ticket tracker
Report bugs with Django or Django documentation in our ticket tracker.

다운로드:

오프라인(Django 3.1): HTML | PDF | ePub
Read the Docs 제공.

django

Hosting by
rackspace

Design by
threespot
&
andrevv

Getting Help

언어: **ko**

문서 버전: **3.1**