

Django 2

파이썬 웹 프레임워크



사용자 관리와 인증



사용자 계정 관리와 인증

- 사용자(User) 계정관리

- 웹 Application을 사용하는 고객/회원의 정보를 관리
- Django는 User를 관리(**등록/수정/탈퇴/정보조회**) 기능을 제공한다.

- 사용자 인증

- 사용자가 웹 Application을 사용할 수 있는 자격이 있는지 확인 하는 절차로 **로그인/로그아웃** 기능을 말한다.
- Django는 사용자 인증을 위한 로그인처리(**login/logout/login 여부확인**) 기능을 제공한다.

사용자 계정관리 구현을 위한 두가지 방법

▪ 구현방식

- 기본 사용자 관리 모델을 이용해 구현

- `django.db.models.User`, `django.contrib.auth`의 Form들을 그대로 이용해 가입/수정/조회/탈퇴를 구현한다.
- 사용자 정보 중 **username(id)**과 **password** 속성만 관리 된다.

- 확장 사용자 관리 모델을 구현

- `username`, `password` 뿐 아니라 추가적인 사용자 정보를 관리해야 할 경우 기본 사용자 관리모델을 확장해서 구현한다.

User Model

- User Model 컴포넌트

- django.contrib.auth.models.**User**
- <https://docs.djangoproject.com/en/5.0/ref/contrib/auth/#user-model>

- User모델의 주요 Field

- **username**: 계정이름(ID)
- **first_name** : 이름
- **last_name**: 성
- **email**: 이메일 주소
- **password**: 패스워드
- **is_staff**: staff여부. True이면 admin 사이트에 로그인할 수 있다.
- **is_active**: 계정 활성화 여부(탈퇴시 False)
- **is_superuser**: True이면 모든 권한을 다 가지게 된다.
- **last_login**: 마지막으로 로그인한 일시
- **date_joined**: 계정 생성 일시



기본 사용자 관리 모델 이용한 구현



- username과 password만 관리

사용자 관리를 위한 ModelForm

- 가입

- `django.contrib.auth.forms.UserCreationForm`
- `username`, `password1`, `password2` 입력을 받아 `username`과 `password` 를 등록한다.

- 정보수정

- `django.contrib.auth.forms.UserChangeForm`

- Password 변경

- `django.contrib.auth.forms.PasswordChangeForm`
- 패스워드는 보안을 위해 암호화 하여 저장되기 때문에 패스워드 변경은 정보 수정과 다른 process로 진행한다.

사용자 관리를 위한 Generic View

- 가입
 - `django.views.generic.CreateView`
- 수정
 - `django.views.generic.UpdateView`
- 정보조회
 - `django.views.generic.DetailView`
- 탈퇴
 - `django.views.generic.DeleteView`
- 비밀번호 변경
 - `django.contrib.auth.views.PasswordChangeView`



확장 사용자 관리 모델 이용한 구현



확장 User 모델

- 기본 User 모델을 프로젝트에 맞게 변경하기.
 - Django는 사용자 관리를 위한 기본 관리 모델을 제공한다. 기본 모델을 사용하면 사용자 등록(가입), 인증(로그인/로그아웃)을 쉽게 구현할 수 있다. 그러나 관리대상 사용자 정보는 username(id)와 password로 한정된다. 프로젝트마다 관리해야 하는 사용자 정보는 다르기 때문에 Django는 프로젝트에 맞게 구현할 수 있는 방법을 제공한다.
- User 모델 확장의 4가지 방법
 1. **AbstractUser**: 기본 사용자 모델을 상속 받아 새로운 필드들을 추가한다.
 2. **AbstractBaseUser**: 기본 사용자 모델을 사용하지 않고 새로운 User 모델을 정의 한다.
(필드들과 기능들을 모두 새로 정의)
 3. **OneToOne 모델**: 기본 User모델과 1 : 1 로 연결되는 자식 모델(테이블)을 만들어 새 필드들을 추가.
 4. **Proxy 모델**: User 모델의 스키마는 그대로 사용하고 기능을 변경/추가

AbstractUser Model을 이용한 확장 User 모델 구현

- 구현 패턴

- Model 클래스 확장

- AbstractUser 를 상속받은 Model 클래스 작성
 - **settings.py 에 User 모델 설정**
 - AUTH_USER_MODEL 변수에 문자열로 등록한다.
 - ex) AUTH_USER_MODEL = 'account.CustomUser'
 - **admin.py에 UserAdmin 구현**
 - UserAdmin
 - 관리자 앱에서 데이터 관리를 위한 화면 구성을 설정하는 클래스
 - 관리자 앱에서 추가한 항목들을 관리하기 위해 구현한다.

- Form 클래스 확장

- 구현한 Model 클래스에 맞게 UserCreationForm을 상속받아 구현한다.
 - UserCreationForm 을 상속받은 Form 클래스 작성
 - 사용자 정의 AbstractUser 모델에 추가한 항목들이 나오도록 구현한다.

AbstractUser Model 클래스 작성

- 사용자 정의 User model 클래스

- models.py 에 구현한다.
- django.contrib.auth.models.**AbstractUser** 를 상속받아 구현
- username, password를 제외한 필드들 추가.

```
from django.contrib.auth.models import AbstractUser
class CustomUser (AbstractUser):
    필드 정의
```

- settings.py 에 구현한 모델 클래스 등록

- AUTH_USER_MODEL 변수에 구현한 AbstractUser 모델클래스를 등록한다.

```
AUTH_USER_MODEL = "accounts.CustomUser"
```

AbstractUser Model 클래스 작성

- admin.py 에 등록
 - admin 앱 에서 사용자정보를 관리하기 위해서 등록
 - **admin.site.register(CustomUser)**
 - 기본 ModelAdmin인 UserAdmin을 재정의
 - Admin 앱에서 사용자 정보 변경/등록을 위한 화면 Customizing 한다.
 - **admin.site.register(CustomUser, CustomUserAdmin)**
 - 변경된 사용자 정보에 맞는 UserAdmin 구현
 - 관리자 페이지에서 사용자 추가/수정을 위한 화면 커스터마이징.
 - `django.contrib.auth.admin.UserAdmin` 를 상속받아 만든다.
 - **설정 항목**
 - **list_display**: Model 속성중 목록에 나올 항목 설정.
 - **add_fieldsets**: 데이터 추가 시 나올 항목 설정.
 - **fieldsets**: 데이터 수정 시 나올 항목 설정

필드셋 구조

```
(  
    "fieldset title", {"fields": (필드명, ..)}  
)
```

UserCreationForm 커스터마이징

- 추가한 Field들이 입력폼에 나오도록 UserCreationForm(**ModelForm**)을 재정의.
- forms.py 에 UserCreationForm을 상속받아 만든다.
 - Meta 내부 클래스도 UserCreationForm.Meta 를 상속받아 구현한다.

```
class CustomUserCreationForm(UserCreationForm):
```

```
    class Meta(UserCreationForm.Meta):
```

```
        model = CustomUser
```

```
        fields = ['username', 'name', 'email', 'birthday']
```



사용자 인증



로그인 구현

- 로그인 **ModelForm**
 - `django.contrib.auth.forms`.**AuthenticationForm**

View – 로그인/로그아웃 처리

- 함수 기반 View

- 로그인 처리 함수

- `django.contrib.auth.login(request, User모델)`
 - `username/password` 가 맞으면 위 함수를 호출하여 로그인 처리한다.

- 로그아웃 처리 함수

- `django.contrib.auth.logout(request)`

- 클래스 기반 View

- **LoginView**

- 로그인 처리
 - GET 요청: 로그인 Form으로 이동.
 - POST 요청: 로그인 처리

- **LogoutView**

- 로그아웃 처리
 - Django 5.0 부터는 POST 방식 요청만 처리한다.

settings.py 로그인 관련 설정

- LOGIN_URL
 - @login_required View를 로그인 하지 않은 사용자가 요청했을 때 이동할 url 설정
 - 기본 url은 **/accounts/login/**
- LOGIN_REDIRECT_URL
 - 로그인 성공 후 이동할 URL 설정
 - LoginView 사용시 설정한다.
- LOGOUT_REDIRECT_URL
 - 로그아웃 후 이동할 URL 설정
 - LogoutView 사용시 설정한다.

로그인 여부 확인

- 로그인을 한 사용자만 요청할 수 있는 View

- django.contrib.auth.decorators.**login_required** 데코레이터
 - View 실행 전 로그인 여부를 체크하는 데코레이터
- 함수 기반 View
 - **@login_required** decorator를 View함수에 추가한다.
- 클래스 기반 View
 - @method_decorator를 이용해 @login_required 데코레이터를 **dispatch** 메소드에 설정한다.
 - class 선언부에 정의한다.

- View에서 로그인한 사용자 User Model 객체 조회

- django.contrib.auth.**get_user** 함수 이용
 - **get_user(self.request)** : 매개변수로 request를 전달하면 로그인한 user의 Model을 반환한다.

- Template 에서 로그인 여부 확인 및 현재 사용자 정보 확인

- **user** 변수를 이용해 사용자의 정보를 사용할 수 있다.
 - 현재 사용자의 User모델을 사용할 수 있다.
 - **로그인 한 경우** 그 사용자의 User 모델객체를, **로그인 하지 않은 경우** AnonymousUser 모델객체 반환.
- user.is_authenticated
 - 현재 사용자가 로그인 했는지 여부를 Boolean 값으로 반환



세션 관리



1) 쿠키

2) 세션

Session관리란?

- Session

- 클라이언트가 하나의 작업을 요청해서 그 작업이 끝날 때까지의 단위를 Session이라고 한다.
하나의 세션 동안 여러 번의 요청과 응답이 반복 될 수 있다.

- Session관리

- 하나의 Session동안 사용자의 데이터가 유지되도록 관리하는 것.
- Http Protocol의 Stateless한 특징의 해결책

- Session 동안 데이터를 유지하는 방법

- 서버에 저장
- 클라이언트(웹브라우저)에 저장

Cookie를 이용한 Session관리

■ 쿠키

- 클라이언트의 정보를 유지하기 위해 서버가 브라우저(client)로 전송하는 text 데이터로 key-value 형태로 관리된다.
- 쿠키는 사이트 별로 관리되며 만료기간이 지나면 삭제 된다.
 - 기본 만료기간: Web Browser가 실행되는 동안.
- 흐름
 - 서버 프로그램에서 다음 요청에서 사용해야 하는 클라이언트의 데이터를 응답할 때 cookie로 보내준다.
 - Web browser는 받은 cookie 데이터를 저장한다.
 - Web browser가 서버에 요청할 때마다 그 site에서 받은 쿠키정보를 Http 요청정보에 담아서 전송한다.

■ 장점

- 서버의 부하를 줄인다.
 - 장기간 저장하는 것이 가능하다.
 - Web Browser나 서버가 종료된 이후에도 보관이 가능하다.

■ 단점

- 관리할 수 있는 데이터의 종류, 크기 제약, 보안상 취약

Cookie 사용

- 클라이언트에게 전달하기
 - HttpResponse의 set_cookie() 메소드를 통해 설정한다.
 - set_cookie(key, value, ..)
 - https://docs.djangoproject.com/en/3.0/ref/request-response/#django.http.HttpResponse.set_cookie
- 클라이언트로 부터 전송된 쿠키 사용하기
 - HttpRequest의 COOKIE 를 통해 사용한다.
 - COOKIE는 딕셔너리 타입으로 **쿠키이름: 쿠키값** 형태
 - Template에서는 request.COOKIES 로 사용한다.

```
def cookie_setting(request):  
    ...  
    response = render(request, '...')  
    response.set_cookie('key', 'value')  
    return response
```

```
def cookie_getting(request):  
    cookie_value = request.COOKIES['key']  
    ...
```

서버에 저장하는 방식

- 유지하려는 Session 데이터를 클라이언트 별로 서버에 저장한다.
 - 유지하려는 session 데이터를 저장하기 위한 공간을 각각의 클라이언트 별로 서버에 만든다.
 - 각각의 공간을 구분하기 위한(클라이언트 구분) Session ID를 생성한다.
 - Session ID를 클라이언트와 서버가 공유한다. 클라이언트는 Session ID를 Cookie로 받아 관리한다.
 - 장고는 요청한 사용자의 **session 데이터를 처음 저장할 때** Session 공간을 만들고 응답 시 Session ID를 Cookie로 전달한다.
- 세션의 장점
 - 클라이언트의 정보를 서버에 저장하기 때문에 쿠키보다 보안에 우수하다.
- 세션의 단점
 - 쿠키보다 보안 면에서 우수하지만 사용자가 많아지면 서버 메모리를 많이 차지하게 되어 성능 저하의 요인이 된다.

서버 저장 방식

- `HttpRequest.session`
 - 딕셔너리 형태
 - 세션값 저장
 - `request.session['key'] = VALUE`
 - 세션값 조회
 - `request.session['key']`
 - `request.session.get('key')`
 - Template에서 조회
 - `{{request.session.key}}`



static 파일 다루기



static 파일과 media 파일

▪ Static 파일

- HTML 문서를 구성하는 정적인 파일들
- 이미지 파일, CSS 파일, Javascript 파일
- 앱이나 프로젝트 단위로 저장하고 사용한다.

▪ Media 파일

- 파일 업로드를 통해 저장된 모든 파일
 - **FileField**나 **ImageField** 를 이용해 저장된 파일
- Database에는 저장된 경로를 저장하고 파일은 파일 스토리지에 저장한다.
- 프로젝트 단위로 저장하고 사용한다.

static 저장을 위한 설정

- 각각의 App에서 사용할 static 파일을 저장경로
 - `app/static/app이름` 경로에 위치 시킨다.
- 프로젝트 전체에서 공통으로 사용되는 static 파일 저장 경로
 - `settings.STATICFILES_DIRS` 에 설정한 경로에 둔다.
- 여러 디렉토리들에 저장된 static 파일들은 `collectstatic` 명령을 사용해 `settings.STATIC_ROOT` 에 설정한 경로에 모은 뒤 서비스에서 사용한다.
- `template`에서 static 파일들 요청 시 사용할 URL 시작 경로는 `settings.STATIC_URL`에 설정된 경로를 사용한다.

static 파일 저장을 위한 절차

- 각 app 별로 app/static/app 디렉토리를 만들고 그 app을 위한 image, css, javascript 파일들을 저장한다. 필요하다면 하위 디렉토리를 만들 수 있다.
- settings.py 에 설정할 내용
 - **STATIC_URL**
 - client에서 static 파일을 요청할 때 사용할 시작 URL을 설정한다.
 - **STATIC_ROOT**
 - 프로젝트에 app별로 여러 군데 나뉘어져 저장한 static 파일들을 모아서 저장할 디렉토리를 만든 뒤, 그 경로를 설정
 - **STATICFILES_DIRS**
 - 프로젝트 전반적으로 사용되는(여러 app들이 공통적으로 사용할) static 파일들이나, root 템플릿에서 사용할 것들을 저장할 디렉토리를 만든 뒤, 그 경로를 설정
 - 리스트에 경로를 지정하므로 여러 디렉토리를 지정할 수 있다.
- python manage.py collectstatic
 - STATIC_ROOT에 static 파일들을 모은다. (복사)

settings.py 에 설정하는 예

```
STATIC_URL = '/static/'  
STATIC_ROOT = os.path.join(BASE_DIR, 'static')  
STATICFILES_DIRS = [  
    os.path.join(BASE_DIR, "config", "static"),  
]
```

template에서 static URL 경로 지정

- static 경로에 저장된 파일들을 template에 지정하는 경우 하드코딩 또는 template 태그 static를 이용해 지정할 수 있다.
 - {% static "STATIC_URL 이후 경로" %}
- 기본 경로는 settings.STATIC_URL 에 설정한 경로로 시작한다.
 - 하드코딩

static_root/**board/img/title.jpg** 파일의 경우

```
<img src = '/static/board/img/title.jpg'>
```

- template 태그 static 이용

```
{% load static %}
```

```
<img src = '{% static "board/img/title.jpg" %}'>
```



파일 업로드와 media 설정



settings.py에 media 경로 설정

- Media 파일

- <https://docs.djangoproject.com/en/5.0/topics/files/>
- Model 클래스의 **FileField** / **ImageField**를 이용해 저장된 모든 파일
- 사용자(client)가 업로드한 파일을 파일로 **파일은 파일 스토리지에 저장하고 DB에는 그 경로를 저장한다.**
- settings.py에 다음 두가지를 설정한다.

- MEDIA_ROOT

- 업로드 된 파일이 저장될 디렉토리 설정

- MEDIA_URL

- 업로드 된 파일을 클라이언트가 요청할 때 사용할 시작 URL (URL Prefix)

```
...  
MEDIA_URL = '/media/'  
  
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')  
...
```

Model의 FileField와 ImageField

- FileField
 - 일반 파일을 저장하기 위한 Field
- ImageField
 - 이미지 파일을 저장하기 위한 Field
 - Pillow 라이브러리를 이용해 이미지의 width/height 정보를 얻는다.
 - ImageField를 사용하기 위해서는 **pillow** 라이브러리가 설치되어 있어야 한다.

```
pip install pillow  
conda install pillow
```

Model의 FileField와 ImageField 주요 속성(옵션)

- upload_to

- 파일을 저장할 settings.MEDIA_ROOT 하위디렉토리 명
- 생략시 settings.MEDIA_ROOT 경로 밑에 저장 한다.
- 같은 이름의 파일이 저장될 경우 뒤에 번호를 붙여 덮어쓰기를 방지한다.
- 경로에 날짜/시간관련 변환문자(%Y, %m, %d, %H, %M, %S) 를 이용해 디렉토리를 동적으로 만들 수 있다.

```
class Item(models.Model):  
    name = models.CharField(max_length=100)  
    ...  
    photo = models.ImageField(upload_to='images')  
    manual = models.FileField(upload_to='docs/%Y/%m/%d')  
    ...
```

- models.ImageField(upload_to='images')
 - MEDIA_ROOT/**images** 아래 저장한다.
- models.FileField(upload_to='docs/%Y/%m/%d')
 - doc/2010/01/05 경로에 저장한다. 경로를 날짜를 이용해 생성

Template에서 MEDIA url 처리

- settings.py의 MEDIA_URL 과 Field의 upload_to 의 경로를 조합해서 url 경로를 만든다. 하드코딩 할 경우 경로가 바뀌면 소스코드를 다 변경해야 한다.

```
MEDIA_URL = '/media/'
```

```
photo = models.ImageField(upload_to='images') => 파일명: my_photo.jpg
```

```
manual = models.FileField(upload_to='docs/%Y/%m/%d') => 파일명: my_file.doc
```

```
<img src='/media/images/my_photo.jpg'>
```

```
<a href='/media/docs/2010/01/05/my_file.doc'>{{item.manual}}</a>
```

- 필드의 url 속성을 이용한 경로 조회

```
<img src = "{{ item.photo.url }}">
```

```
<a href='{{item.manual.url}}'>{{item.manual}}</a>
```

개발 서버를 위해 media 파일 url 등록

- static 파일과 다르게 장고 개발 서버는 Media 파일을 클라이언트에게 서비스 하는 것을 지원하지 않는다.
- 개발 서버에서 media 파일 서비스를 위해 다음을 config/urls.py 에 등록 한다.

```
.....
```

```
from django.conf.urls.static import static
```

```
from config import settings
```

```
urlpatterns = [...]
```

```
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

파일 업로드 - HTML

- form 설정
 - method: post
 - enctype: multipart/form-data
- input 태그
 - type: file

```
<form action="..." method="post" enctype="multipart/form-data">  
...  
  <input type="file" name="upfile">  
...  
</form>
```