

Django - Auth1

# Session 11

---

NEXT X LIKELION 이혁준

# Foreign Key 복습

## RDBMS (Relational DBMS)

- Foreign Key

[ 고려대학교 학생 ]

학번	이름	학과	수강 강의번호
2021130840	손예원	컴퓨터학과	4
328956274	최유빈	미디어학부	1
1234567890	김태균	체육교육과	2
2037561298	나성희	경영학과	3
3042783569	박소정	경영학과	3

참조

[ 고려대학교 강의 ]

강의번호	강의명	담당교수
1	미디어경제	김정현
2	보건교육론	이승희
3	회계원리	김진배
4	운영체제	유헌창

### Foreign Key :

참조 대상 테이블의 tuple 튜플(=행)을 식별할 수 있는 key  
[학생]과 [강의] 테이블은 각각 존재하며, 두 테이블 간에는 참조 관계가 있다

NEXT X LIKELION

출처 : Session 8. 예원이~

## 1:N 구조

```
class Post(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField()

    def __str__(self):
        return self.title

class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')
    content = models.TextField()

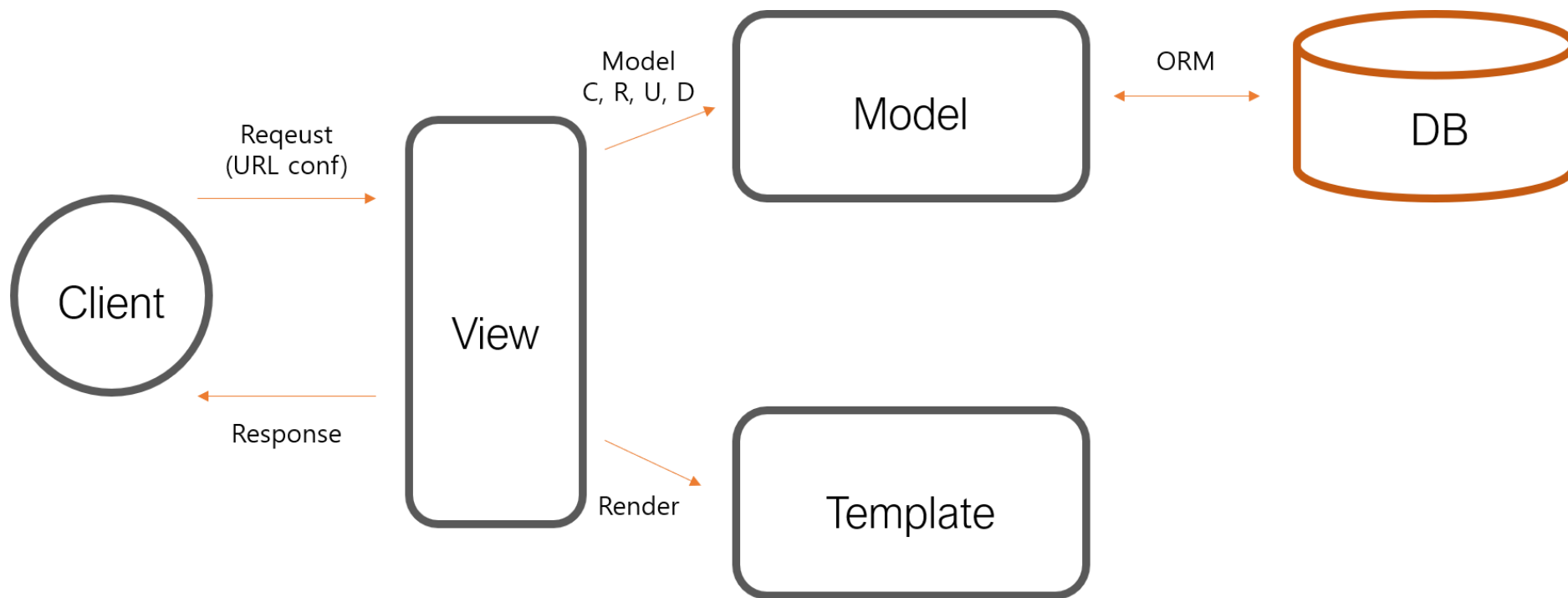
    def __str__(self):
        return self.content
```

# MTV Pattern : 결국 데이터를 처리하는 로직을 정리한 구조

Model: 데이터의 흐름을 규정하여 규격을 맞추는 것

Template: 유저에게 데이터를 어떻게 보여주고, 어떻게 받을지를 규정하는 것

View: 규정된 모델을 토대로 데이터의 입출력을 이용해 비즈니스 로직을 실행하는 것



# 장고 템플릿 문법

- `{% extends 'base.html' %}`

Base.html 에 틀을 짜고

나머지 자식 템플릿에서는 이 base.html을 상속합니다!

- `{% block 태그이름 %}`

페이지별로 바뀌는 부분은 block으로 감싸서

- `{% endblock 태그이름 %}`

자식 템플릿에서 내용을 바꿔 줍니다.

# 목차

## 0. Authentication(인증), Authorization(인가)

인증과 권한에 대해서

DJANGO library - `django.contrib.auth`

decorator

## 1. Authentication(인증)

회원가입/로그인/로그아웃


## 2. Authorization(인가)


로그인 여부에 따라 다르게 보이는 navbar, 접근 가능한 페이지


글 작성자만 수정, 삭제 권한 부여

# 인증(Authentication)???


어떤 유저가 해당 유저가 주장하는 그 유저가 맞는지 확인하는 것

 @natureundersea3615 · 3개월 전  
안녕 하니아.. 댓글 보면 좋아요 눌러줘

👍 842 💬 

 @P.\_\_. · 2개월 전  
나 하니인데 개추 눌렀다

👍 2.2천 💬

 @natureundersea3615 · 2개월 전  
[@P.\\_\\_.](#) 하니는 그런말투 안써요

👍 1.1천 💬

# 인가(Authorization)???

해당 유저, 또는 유저가 속한 집단에 따라 서비스에 대한 권한을 부여하거나 거부하는 것



이거



아님 이거



# Authetication + Authorization

보안은 웹 서버의 생명



# Django.contrib.auth

Django의 보안 library - 추상화

<https://docs.djangoproject.com/en/5.0/ref/contrib/auth/>

Django에서는 개발 부담을 덜어주기 위해서 백엔드에서 사용하는 매우 많은 부분을 개발자가 그대로 쓸 수 있도록 라이브러리 형태로 제공하고 있음

이중 django.contrib.auth에서는 보안 관련된 많은 기능들을 제공함. Login, authenticate, login\_required 등을 제공함.(물론 이는 양날의 검으로 작용함)

자세한 사항은 documentation을 확인하자! 개발자는 평생 공부하는 직업...

# Django.contrib.auth

Django의 보안 library - 설명

<https://docs.djangoproject.com/en/5.0/ref/contrib/auth/>

1. AbstractBaseUser, PermissionsMixin : User Model을 만들기 위한 기반!

```
class User(AbstractBaseUser, PermissionsMixin):
    objects = UserManager()

    username = models.CharField(max_length=255, unique=True)
    is_active = models.BooleanField(default=True)

    USERNAME_FIELD = "username"
    REQUIRED_FIELDS = ["username"]

    def __str__(self) -> str:
        return self.username
```

# Django.contrib.auth

Django의 보안 library - 설명

<https://docs.djangoproject.com/en/5.0/ref/contrib/auth/>

1. AbstractBaseUser, PermissionsMixin : User Model을 만들기 위한 기반!

## Custom users and permissions

To make it easy to include Django's permission framework into your own user class, Django provides **PermissionsMixin**. This is an abstract model you can include in the class hierarchy for your user model, giving you all the methods and database fields necessary to support Django's permission model.

<https://docs.djangoproject.com/en/5.0/topics/auth/customizing/#django.contrib.auth.models.AbstractBaseUser>

# Django.contrib.auth

## Django의 보안 library - 설명

2. login\_required decorator - 자동으로 로그인을 검증해주는 decorator

<https://docs.djangoproject.com/en/5.0/topics/auth/default/#the-login-required-decorator>

```
@login_required
def new(request):
    if request.method == "POST":
        title = request.POST["title"]
        content = request.POST["content"]

        new_post = Post.objects.create(title=title, content=content)
        return redirect("detail", new_post.pk)

    return render(request, "new.html")
```

# Django.contrib.auth

코드의 효율적 구조 - Decorator

네?? Decorator요?

이거 -> 

```
@login_required  
def new(request):
```

# Django.contrib.auth

## 코드의 효율적 구조 - Decorator

### decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for `decorators` are `classmethod()` and `staticmethod()`.

The `decorator` syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

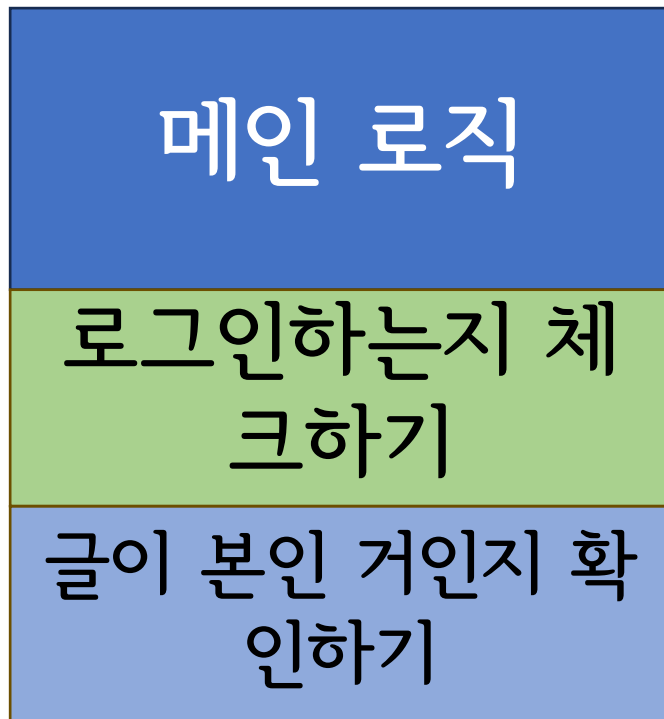
```
def f(arg):  
    ...  
f = staticmethod(f)  
  
@staticmethod  
def f(arg):  
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for [function definitions](#) and [class definitions](#) for more about `decorators`.

# Django.contrib.auth

## 직관적 구조

### 게시물 관련 View



⋮

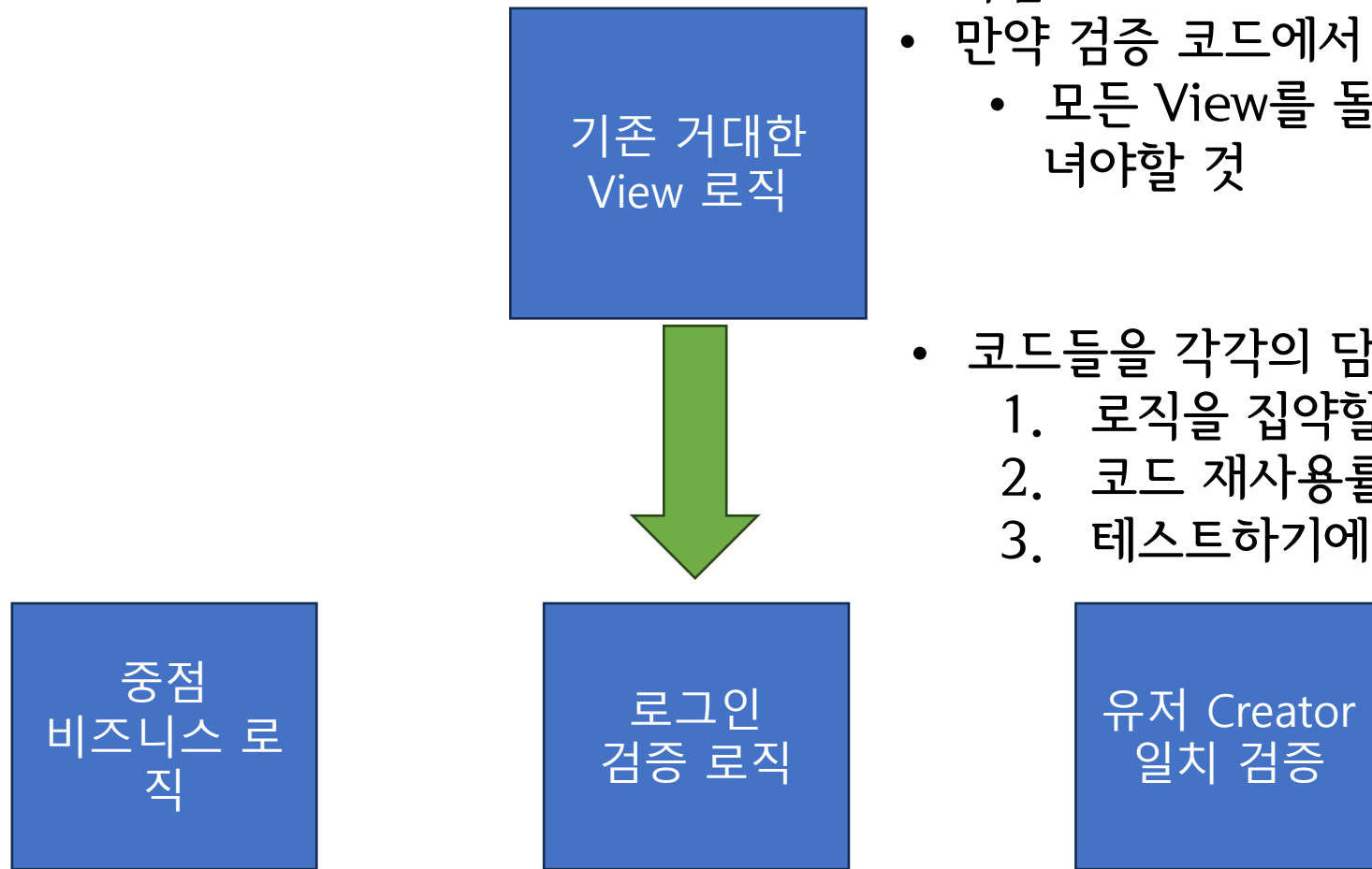
1. 나는 너가 게시물에 대해서 title, content를 유저에게 보여주고, 수정할 수 있도록 해줬으면 좋겠어!
2. 아 근데 생각해보니까 아무나 글을 보는 건 좀 그러니까, 로그인한 사람만 보게 하자
3. 아 근데 수정도 본인 글만 하게 해야겠는데?? 남의 글을 수정하면 안 되겠다
4. 와 근데 댓글도 그러면 본인만 수정, 삭제하도록 해야겠네?? 코드를 거기에도 추가해야하나.. 또 다른 곳에도 추가해야하면 어떡하지?

로직에만 집중하고 싶은데, 다른 문제들이 떠오르네..



# Django.contrib.auth

직관적 구조 : 관심사의 분리



- 기존의 거대한 View 로직에서는 비즈니스 로직에 집중하기 보다는 다른 검증 로직 등이 추가되며 코드가 길어지고 중복됨
- 만약 검증 코드에서 수정 사항이 발견되면?
  - 모든 View를 돌아다니면서 중복 코드를 수정하고 다녀야할 것
- 코드들을 각각의 담당 로직에 따라 분리한다면
  1. 로직을 집약할 수 있어 이해도가 올라가고,
  2. 코드 재사용률을 높혀, 중복된 코드를 줄일 수 있음
  3. 테스트하기에 용이함

# Django.contrib.auth

직관적 구조 : decorator란?

Decorator란 함수에 덮어 씌어져 추가적인 기능을 제공하는 함수  
Wrapper function



@login\_required

1. 사용자가 로그인 되어 있는지를 검증하고,
  - 2-1. 안 되어있다면 login\_url로 보내고
  - 2-2. 되어있다면 request.user에 유저 객체를 첨부해서
3. 다음 핵심 비즈니스 로직을 실행한다.

감싸고 있다는 것은, 핵심 비즈니스 로직이 끝난 다음에도 무언가를 수행할 수 있다는 것

왜케 중요하게 다루지? => 오늘 씬

디자인 따위는 신경도 안(못)쓰는  
장고 보안 실습 열차..출발합니다

어려우면 질문 많이 하세요

사실 gpt가 더 잘 앞



# 로그인

## 오늘의 요청사항

### 요청사항

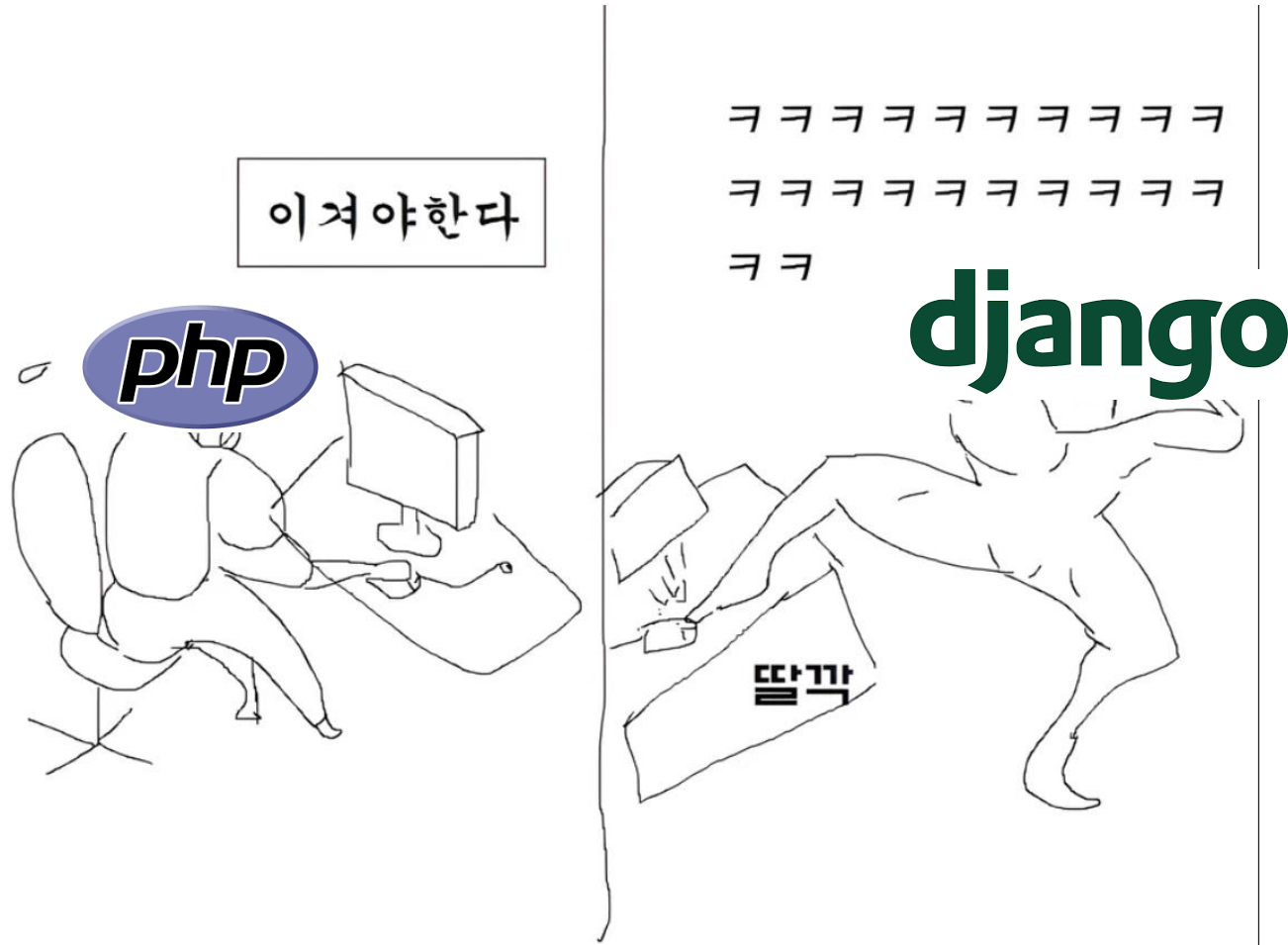
1. 유저가 회원가입을 할 수 있으면 좋겠어요
2. 유저가 로그인, 아웃을 할 수 있으면 좋겠어요
3. 로그인한 사람만 게시물 생성 및 댓글 작성했으면 좋겠어요
4. 본인의 게시물, 댓글만 수정, 삭제할 수 있으면 좋겠어요

django : 어 형이야.



# 로그인

## Authentication 요청사항



# django

1. 유저 객체 Base Model 제공
2. 유저 생성(회원가입) Form 제공 + 검증
3. 로그인 함수 제공
4. 로그아웃 함수 제공
5. 권한 인증 자동화 및 기본 포맷 제공
6. 유저 및 기타 데이터 수정용 어드민 페이지 제공
7. ORM 자체 제공
8. Related entities 자동 제공

# 실습환경 세팅

```
$ mkdir session11
```

```
$ cd ./session11
```

```
$ 압출 풀고 프로젝트 들어가기
```

```
$ pipenv shell
```

```
$ pipenv install django
```

```
$ cd basic_auth
```

```
$ cd code .
```

Blog에는 기존에 다루던 게시물, 댓글 관련 로직이 들어갈 것이고,

Authapp에는 새로 구현할 보안(유저) 관련 로직이 들어갈 것임

=> 관심사의 분리

# | 백엔드 구현 : 이벤트 스토밍

요청 사항

- 1. 유저 객체 생성 및 기존 블로그와 연동
- 2. 보안 로직 구현

비회원

Home

게시물  
열람 하기

회원가입하기

회원

Home

게시물  
열람 하기

로그인  
하기

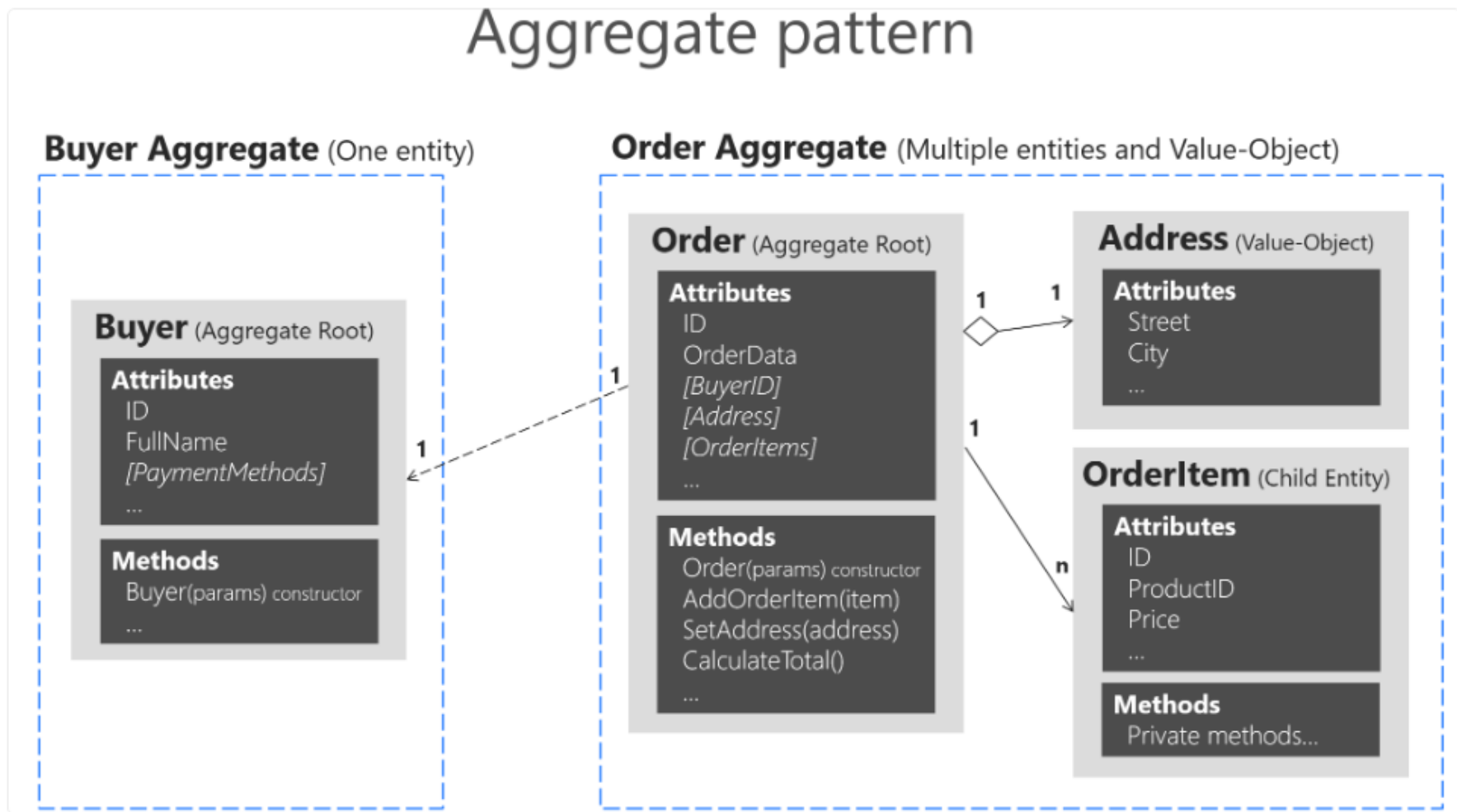
게시물  
생성, 삭제

댓글  
작성, 삭제

# 백엔드 구현 : 도메인 정의

<https://incheol-jung.gitbook.io/docs/q-and-a/architecture/ddd>

## Aggregate pattern





# 백엔드 구현 : 도메인 정의

<https://incheol-jung.gitbook.io/docs/q-and-a/architecture/ddd>

## 1. 유저

- 유저는 다음과 같은 정보를 받아 생성됩니다.
- Nickname : 유저의 닉네임을 담는 중복되지 않는 값입니다.
- Password : 유저의 보안을 담는 비밀번호입니다.

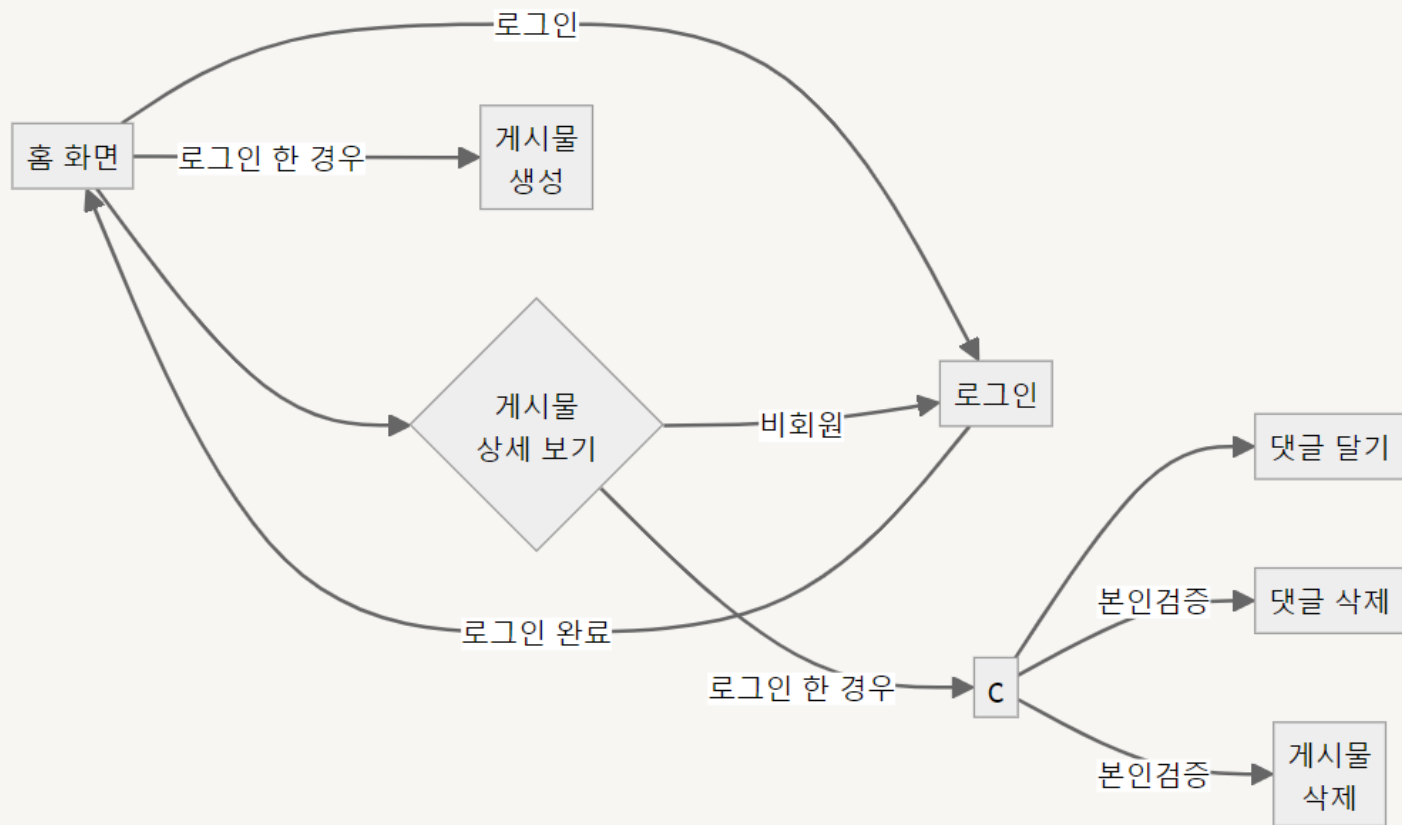
## 2. 글(Post)

- 글은 다음과 같은 정보를 저장해야 합니다.
- Content: 글의 내용을 담고 있는 값입니다.
- Title: 글의 제목을 담고 있는 값입니다.
- Creator: 글을 작성한 User를 담고 있는 값입니다.

## 3. 댓글(comment)

- 댓글은 다음과 같은 정보를 저장해야 합니다.
- Content: 댓글의 내용을 담고 있는 값입니다.
- Post: 댓글이 작성된 Post를 담고 있는 값입니다.
- User: 글을 작성한 User를 담고 있는 값입니다.

# 백엔드 구현 : 로직 설계



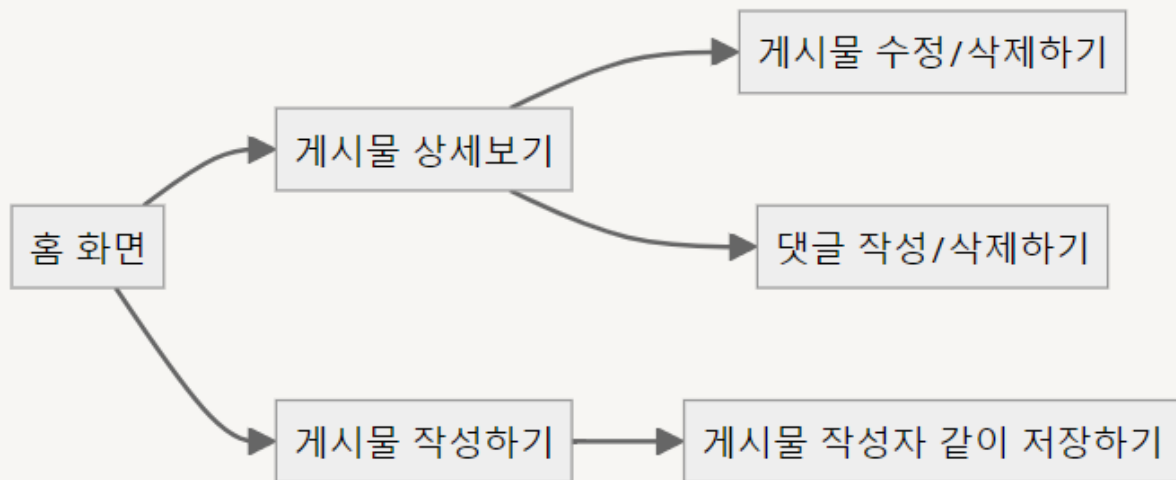
사실 이렇게 뭉터기로 하지는 않고, 도메인 별로 나누어서 진행합니다.

플로우를 명확히 하면서 각 단계를 설계할 수 있습니다.

지금 잠을 잘 못 잤어서 설계를 잘 못하겠네요

여기서는 좀 이상하게 나오지만, 게시물 도메인과, 검증(유저) 도메인으로 나누어서 설계를 하면 좋을 것 같습니다.

# 백엔드 구현 : 로직 설계



로그인 요청 → 검증 및 유저 데이터 전송

서비스 접근 → 권한 확인 및 서비스 제공

데이터 요청 → 생성자 확인 → 권한 확인 및 데이터 제공

좀더 간단하고 직관적이게 바뀌었습니다.

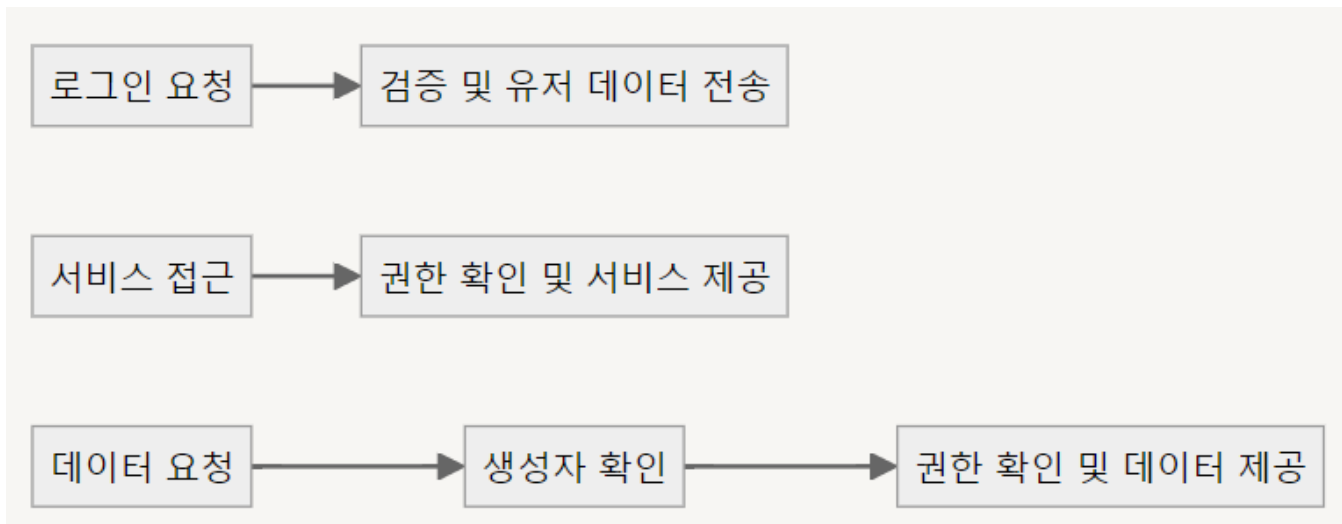
지금이야 서비스 자체가 매우 단순하지만, 이미 구현된 서비스만 2만줄이 넘어가고, 사이드 이펙트까지 고려해야하는 상황이라면 이렇게 플로우 차트를 그려보는 것도 좋습니다.

# 백엔드 구현 : 개발 시작

목차에서 적었듯, 인증인 signup과 login부터 구현하겠습니다.

하지만, 지금 작성자를 저장할 User 객체가 없죠?

User를 먼저 모델을 설계하도록 하겠습니다.



# 백엔드 구현 : AbstractUser

## Using a custom user model when starting a project

If you're starting a new project, it's highly recommended to set up a custom user model, even if the default User model is efficient for you, you can customize it in the future if the need arises:

```
class AbstractUser(AbstractBaseUser, PermissionsMixin):
    """
    An abstract base class implementing a fully featured User model with
    admin-compliant permissions.

    Username and password are required. Other fields are optional.
    """

    username_validator = UnicodeUsernameValidator()

    username = models.CharField(
        _("username"),
        max_length=150,
        unique=True,
        help_text=_(
            "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only."
        ),
        validators=[username_validator],
        error_messages={
            "unique": _("A user with that username already exists."),
        },
    )
    first_name = models.CharField(_("first name"), max_length=150, blank=True)
    last_name = models.CharField(_("last name"), max_length=150, blank=True)
    email = models.EmailField(_("email address"), blank=True)
    is_staff = models.BooleanField(
```

Django는 즉시 사용할 수 있는 AbstractUser라는 모델을 제공합니다.

AbstractUser는 basemanager, usercreationform과 함께 말그대로 유저와 관련된 모든 기능을 이미 완성된채로 개발자에게 제공합니다.

하지만 이 모든 게 장점만은 아닙니다.

## 백엔드 구현 : User



Django가 제공하는 템플릿들은 편의를 제공하지만 동시에 비즈니스 로직의 플로우를 강제합니다.

예를 들어 AbstractUser는 first\_name, last\_name, email의 기재를 강제합니다.

Session data를 이용한 User의 로그인 로그아웃은 개발자의 '편의'를 위해 개발자가 알 수 없는 under the hood에서 로직을 수행합니다

## 백엔드 구현 : User



개발자가 package를 선택할 때는 전혀 예상하지 못했던 문제가 발생할 수 있습니다. Library 또는 외부 솔루션을 사용할 때는 꼭 그 사이드 이펙트를 확인해야 합니다.

그리고 공수 효율을 잘 따져서 가능한 것은 직접 구현하는 것도 정말 좋은 방법 중 하나입니다.

오늘은 그래서 AbstractUser가 아닌, django에서 제공하는 최소한의 템플릿인 AbstractBaseUser를 사용해보겠습니다.

근데 장고는 진짜 친절해서 이것마저도 편리해요

# 백엔드 구현 : User

## 간단하죠?

```
from django.db import models
from django.contrib.auth.models import (
    AbstractBaseUser,
    BaseUserManager,
    PermissionsMixin,
)
```

AbstractBaseUser는 django에서 제공하는 가장 간단한 유저 모델 프리셋입니다.

비밀번호, 유저 로그인 여부 확인, 유저 생성 등에 대해서 기본 기능을 제공합니다. 궁금하신 분들은 AbstractBaseUser 코드를 확인해보시는 것을 추천합니다.

```
class User(AbstractBaseUser, PermissionsMixin):
    objects = UserManager()

    username = models.CharField(max_length=255, unique=True)

    USERNAME_FIELD = "username"

    def __str__(self) -> str:
        return self.username
```

ㅇㅋ ㅇㅋ

근데 objects는 뭐야?



# 백엔드 구현 : objects

```
class User(AbstractBaseUser, PermissionsMixin):
    objects = UserManager()

    username = models.CharField(max_length=255, unique=True)

    USERNAME_FIELD = "username"

    def __str__(self) -> str:
        return self.username
```

```
def delete(request, post_pk):
    post = Post.objects.get(pk=post_pk)
    post.delete()
    return redirect("home")
```

위에 보이는 objects랑 똑같은 친구입니다.

Objects는 데이터베이스와 소통(CRUD)하기 위해 필요한 Database manager입니다.  
ORM이라고 들어보셨나요?

즉 저희가 사용하는 .get .find .update 등의 함수들을 데이터베이스가 이해할 수 있는 명령어로 바꾸어 주는 매니저입니다.

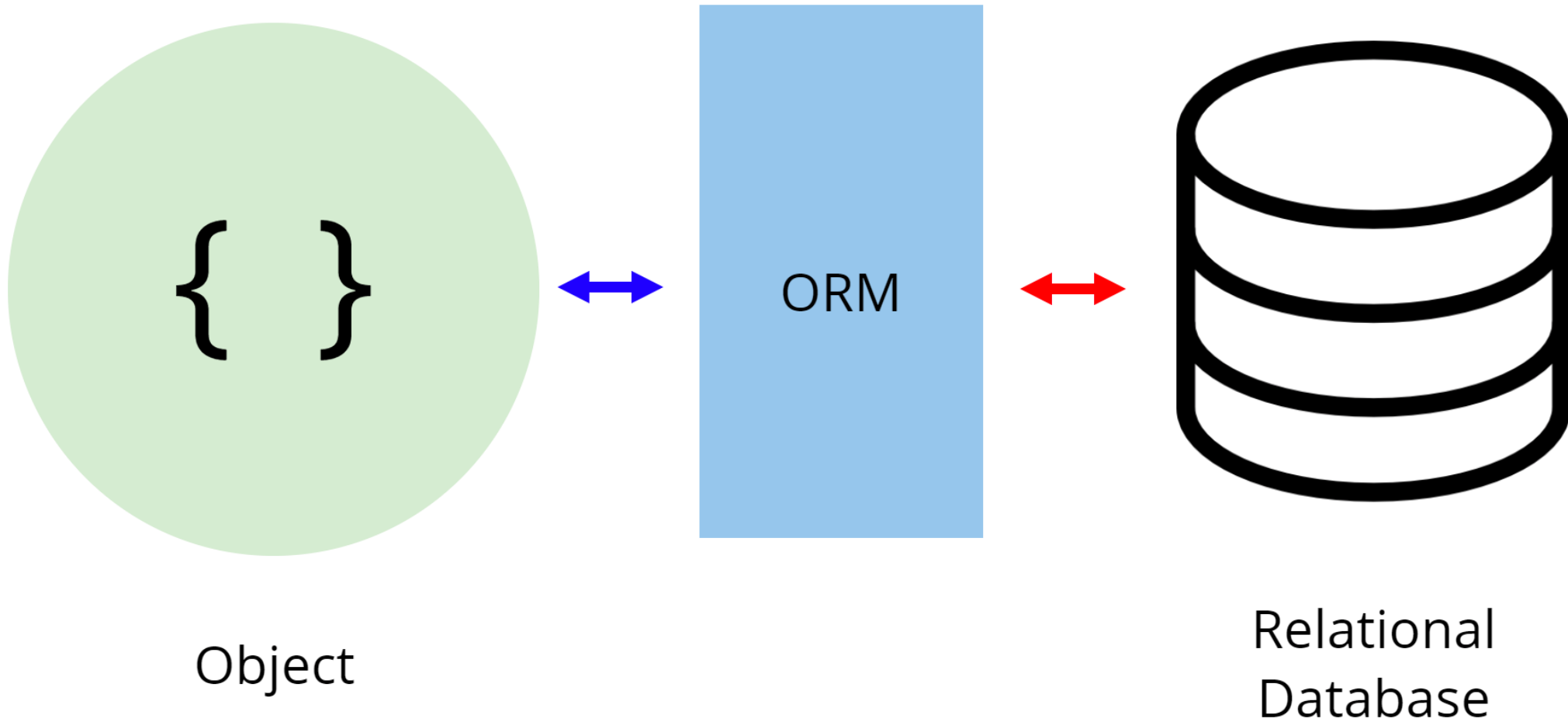
models.Model에서 기본적으로 제공합니다. AbstractBaseUser는 models.Model을 상속했기 때문에, 기본 objects manager를 제공합니다. 하지만, 우리는 특수한 모델인 User를 생성하려 하기 때문에 기본 manager가 제공하는 함수 외에도 추가적인 기능들이 필요하기 때문에 별도의 매니저를 생성해주어야 하는 것입니다.

## 백엔드 구현 : objects



... 그 열차 잠시 갈아타죠

## 백엔드 구현 : ORM



# 백엔드 구현 : ORM

그러니까

User.save(nickname = 'test', password='1234')인 코드를

INSERT INTO "user" ('nickname', 'password')

Values ('test', '1234')

로 변환해주는 친구를 ORM이라고 합니다.

어 근데, password를 그냥 '1234'로 저장해도 될까요..? 위험해 보이죠? 그리고 password가 너무 짧고? 아 그리고 어드민도 설정해줘야 하는데..

User는 이렇듯 이런저런 특수 케이스가 많기 때문에 기존 create보다 더 복잡한 생성 함수가 필요합니다.

# 백엔드 구현 : UserManager

```
class UserManager(BaseUserManager):  
    use_in_migrations = True  
  
    def create_user(self, username, password):  
        if not username:  
            raise ValueError("must have username")  
        if not password:  
            raise ValueError("must have user password")  
  
        user = self.model(username=username)  
        user.set_password(password)  
        user.save(using=self._db)  
        return user  
  
    def create_superuser(self, username, password):  
        user = self.create_user(username=username, password=password)  
        user.is_superuser = True  
        user.save(using=self._db)  
        return user
```

보면 그냥 objects.create이 아닌, User 객체에 대한 검증을 추가하고, set\_password로 password도 해싱해서 저장한 것을 볼 수 있습니다.

그리고 admin 계정도 만들 수 있어야하므로, create\_superuser함수를 추가했습니다.

create\_user는 없어도 괜찮지만(조금 귀찮은 방식으로 구현 가능), create\_superuser는 필수적으로 구현해야 합니다(manage.py 에서 사용하는 함수).

Use\_in\_migrations를 설정함으로써 DB에서 테이블 자체를 변경할 일이 있을 때, 해당 함수를 사용한다는 뜻입니다.

## 유저 : 해치웠나?

아직 남았습니다. User 생성은 했지만 이 유저를 아직 django에서 제공하는 backend authentication에 등록하지 않았습니다.

```
AUTH_USER_MODEL = "authapp.User"
```

Base\_auth/settings.py에 위 코드를 추가해줍니다. 우리가 만든 모델을 기본 유저 모델로 사용하라는 의미입니다.

이를 통해서 django가 기본적으로 제공해주는 모든 보안 서비스를 사용할 수 있는 유저 모델을 생성했습니다.

# 백엔드 구현 : 회원가입

유저를 저장할 수 있는 객체를 생성했으니, 이제는 회원가입에 쓰일 생성 틀을 만들어봅시다. 저번 세션 기억을 되살려보면 우리는 form을 이용해서 html에서 서버에 요청했던 것을 기억할 수 있습니다.

```
<div class="new-post_container">
  <form method="POST" action="">
    {% csrf_token %}
    <input type="text" placeholder="제목을 입력하세요!" name="title" />
    <textarea placeholder="내용을 입력하세요!" name="content"></textarea>
    <input id="new-post_btn" type="submit" value="작성하기" />
  </form>
</div>
```

이번에도 이렇게 form을 직접 생성해서 유저 nickname, password 칸을 만들면 되는 걸까요..? 검증 로직은 view에서 구현하나요? 생성 그 자체는요?? 그걸 다 구현할 필요가 있을까요?

# | 백엔드 구현 : UserCreationForm : forms.py

Django는 이것마저도 자동화를 지원합니다.

**django** : 어 형이야.

Authapp/forms.py 로 이동해주세요



# 백엔드 구현 : UserCreationForm : forms.py

```
class UserCreationForm(forms.ModelForm):
    password1 = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(
        label="Password Confirmation", widget=forms.PasswordInput
    )

    class Meta:
        model = User
        fields = ("username",)

    def clean_password2(self):
        cd = self.cleaned_data
        if cd["password1"] != cd["password2"]:
            raise forms.ValidationError("Passwords don't match.")
        return cd["password2"]

    def save(self, commit=True) -> User:
        user = User.objects.create_user(
            username=self.cleaned_data["username"],
            password=self.cleaned_data["password1"],
        )
        if commit:
            user.save()
        return user
```

Form field 설정하기

어느 모델에 대한 생성 form인지 확인  
어떤 field를 가져올지 확인

검증 로직

모델 생성 코드

# 백엔드 구현 : UserCreationForm : views.py

```
def signup(request):  
    if request.method == "POST":  
        form = UserCreationForm(request.POST)  
        if form.is_valid():  
            user = form.save()  
            django_login(request, user)  
            return redirect("home")  
        else:  
            print(form.errors)  
    else:  
        form = UserCreationForm()  
  
    return render(request, "signup.html", {"form": form})
```

is\_valid(): 유저가 입력한 값이 올바른지 확인  
clean\_custom\_value()도 자동 실행

Form.errors: 유저가 어떤 오류를 행했는지 내역

Form.save(): 유저 생성

Django\_login => login() 장고의 **세션 로그인**

```
<form method="POST">  
    {% csrf_token %}  
    <div class="posts_box">  
        {% if form.errors %}  
        <ul>  
            {% for field, error_list in form.errors.items %}  
            {% for error in error_list %}  
            <li>{{ error }}</li>  
            {% endfor %}  
            {% endfor %}  
        </ul>  
        {% endif %}  
        {{ form }}  
    </div>  
    <div class="registration-box">  
        <button id="post-button">회원가입</button>  
    </div>  
</form>
```

# 백엔드 구현 : LoginForm : forms.py, views.py

```
class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

```
def login(request):
    if request.method == "POST":
        form = LoginForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data["username"]
            password = form.cleaned_data["password"]
            user = authenticate(request, username=username, password=password)
            if user is not None:
                django_login(request, user)
                return redirect("home")
            else:
                form.add_error(None, "Invalid username or password")
        else:
            form = LoginForm()
    return render(request, "login.html", {"form": form})
```

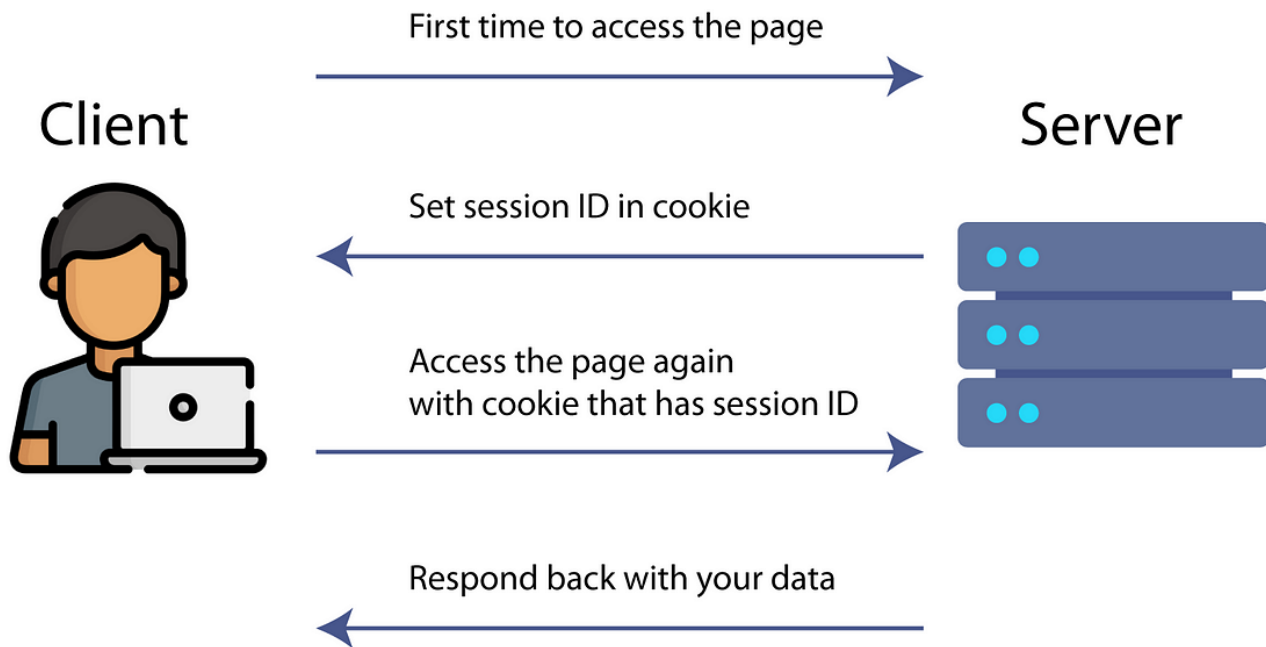
Authenticate: 해당 정보를 가진 유저가 있는지 확인 = 로그인

Django\_login => session login  
Logout => session logout

Is\_authenticated: 유저가 session에 login data를 가지고 있는지 확인

```
def logout_user(request):
    if request.user.is_authenticated:
        logout(request)
        return redirect("home")
    else:
        return redirect("login")
```

# 백엔드 구현 : session이란?



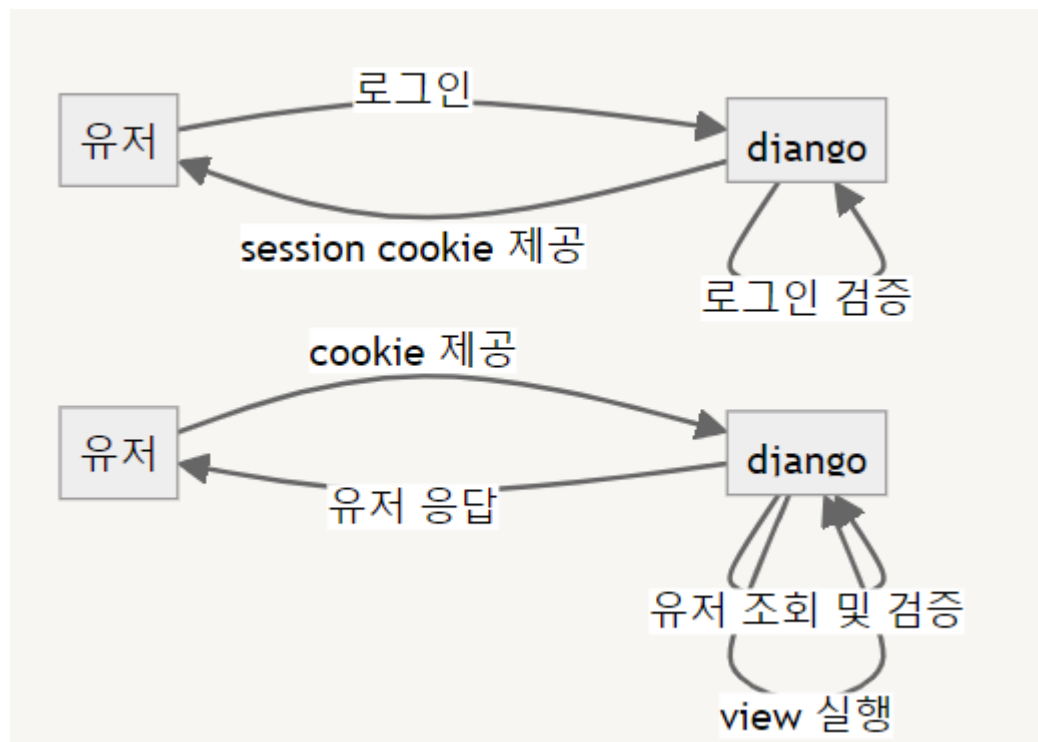
잠깐의 시간동안 정보를 담을 수 있는 공간

Authentication Cookie로 유저의 로그인 정보를 저장합니다.

하지만, 저희 코드에서는 이 데이터를 다루는 코드가 보이지 않습니다.

분명 매 요청마다 보안 쿠키를 같이 넣어서 제출하고, 이를 검증해야 할텐데요.

# 백엔드 구현 : session이란?



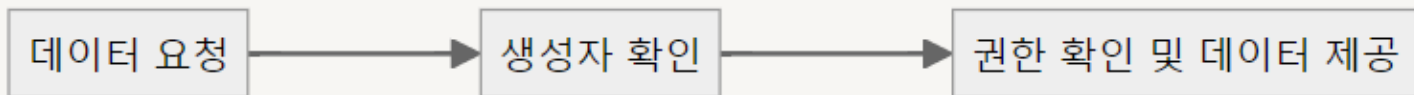
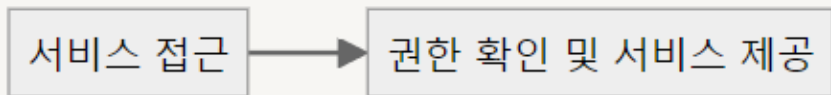
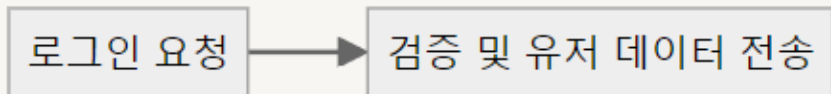
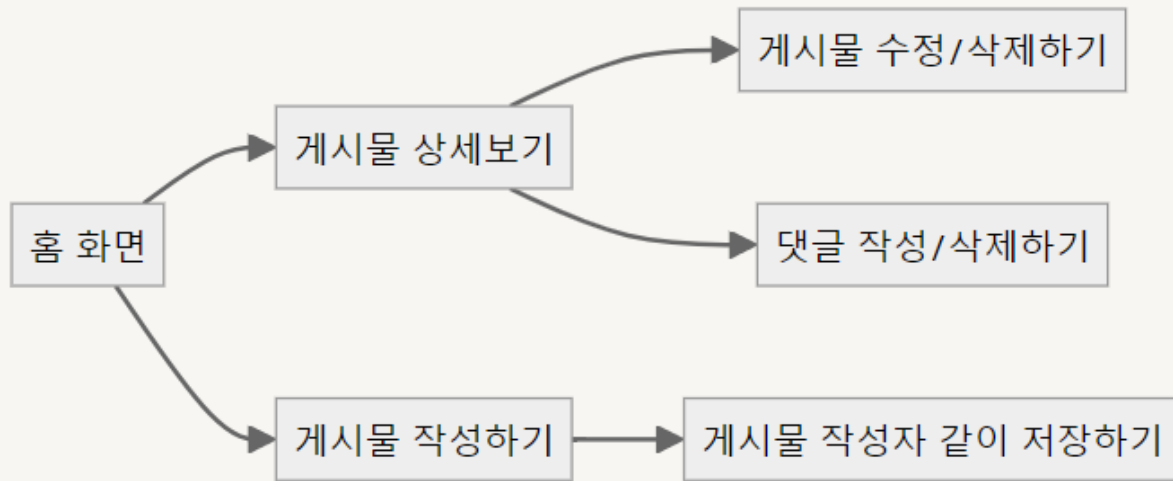
django auth는 유저 로그인, session cookie 제공, session cookie 전송, 쿠키 검증, 유저 조회, 유저 데이터 제공 등 모든 부분을 자동으로 해주기 때문에 저희가 직접 코드로 구현해야 할 필요가 없습니다.

# django

# 잠시 쉬시면서, 질문 좀 받죠



# 백엔드 구현 : 로직 설계



자 이제 인가 관련한 구현 내용들을 보겠습니다.

게시물 삭제 권한에 가기 전에, navbar를 유저 로그인에 따라서 다르게 보이게 하는 것부터 구현하죠

앞에서 지나가듯 말한 `AbstractBaseUser`가 제공하는 `is_user_authenticated` 기억하나요?

# 백엔드 구현 : 유저 로그인 여부에 따른 navbar 변경

New Home 로그인 회원가입

New Home 로그아웃

이렇게 유저의 로그인 여부에 따라서 서로 다른 모습을 보여주기 위해서는 전에 말한 session data가 필요합니다.

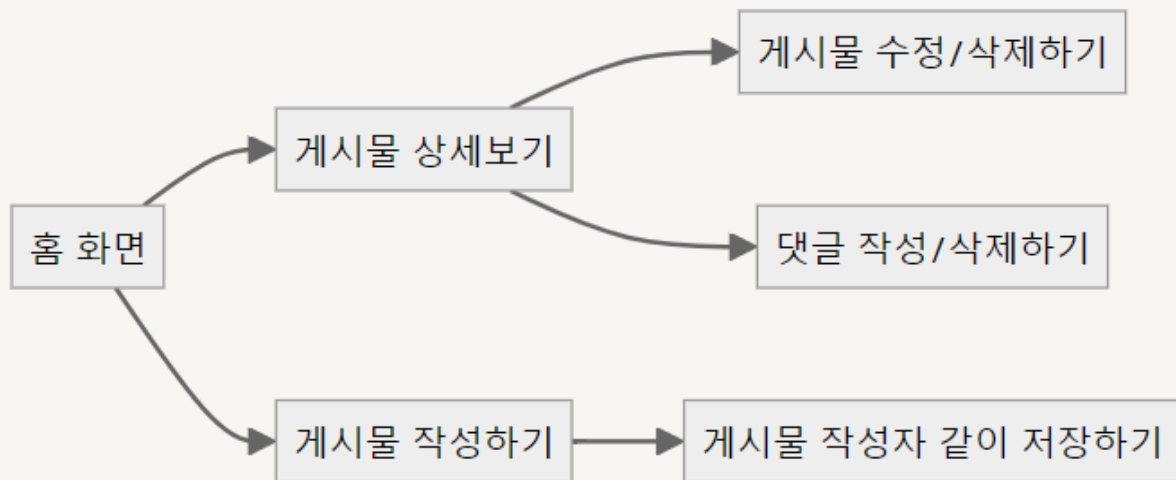
AbstractBaseUser가 제공하는 is\_user\_authenticated는 유저를 대신해서 session data를 확인해서 user의 로그인 여부를 확인합니다.

간단하죠?

```
{% if user.is_authenticated %}
<li class="navbar__menu__item">
  <a href="{% url 'logout' %}">로그아웃</a>
</li>
{% else %}
<li class="navbar__menu__item">
  <a href="{% url 'login' %}">로그인</a>
</li>
<li class="navbar__menu__item">
  <a href="{% url 'signup' %}">회원가입</a>
</li>
{% endif %}
```



# 백엔드 구현 : 로직 설계



자 이제 그러면 다시 게시물을 작성자만 삭제할 수 있도록 하는 것을 구현하겠습니다.

해당 로직을 구현하기 전에 먼저 게시물에 작성자가 누구인지 저장할 수 있도록 수정하죠

로그인 요청 → 검증 및 유저 데이터 전송

서비스 접근 → 권한 확인 및 서비스 제공

데이터 요청 → 생성자 확인 → 권한 확인 및 데이터 제공

## 백엔드 구현 : Post, Comment 모델 변경

이제 Post와 Comment는 자신들을 누가 만들었는지를 설정할 수 있습니다. Foreign Key relation 기억하시죠?

```
class Post(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField()
    creator = models.ForeignKey(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="posts"
    )

    def __str__(self):
        return self.title
```

# 백엔드 구현 : Post, Comment 모델 변경

<https://docs.djangoproject.com/en/5.0/topics/auth/customizing/#auth-custom-user>

Django에서는 User 객체를 직접 호출하는 것을 추천하지 않습니다.  
get\_user\_model()이나 settings.AUTH\_USER\_MODEL을 사용해서 간접적으로 호출하는 것을 추천합니다.

```
get_user_model().objects.filter(username="admin").exists()
```

```
creator = models.ForeignKey(  
    settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="posts"  
)
```

# 백엔드 구현 : 검증 로직 재확인

## 메인 로직

로그인하는지 체크하기

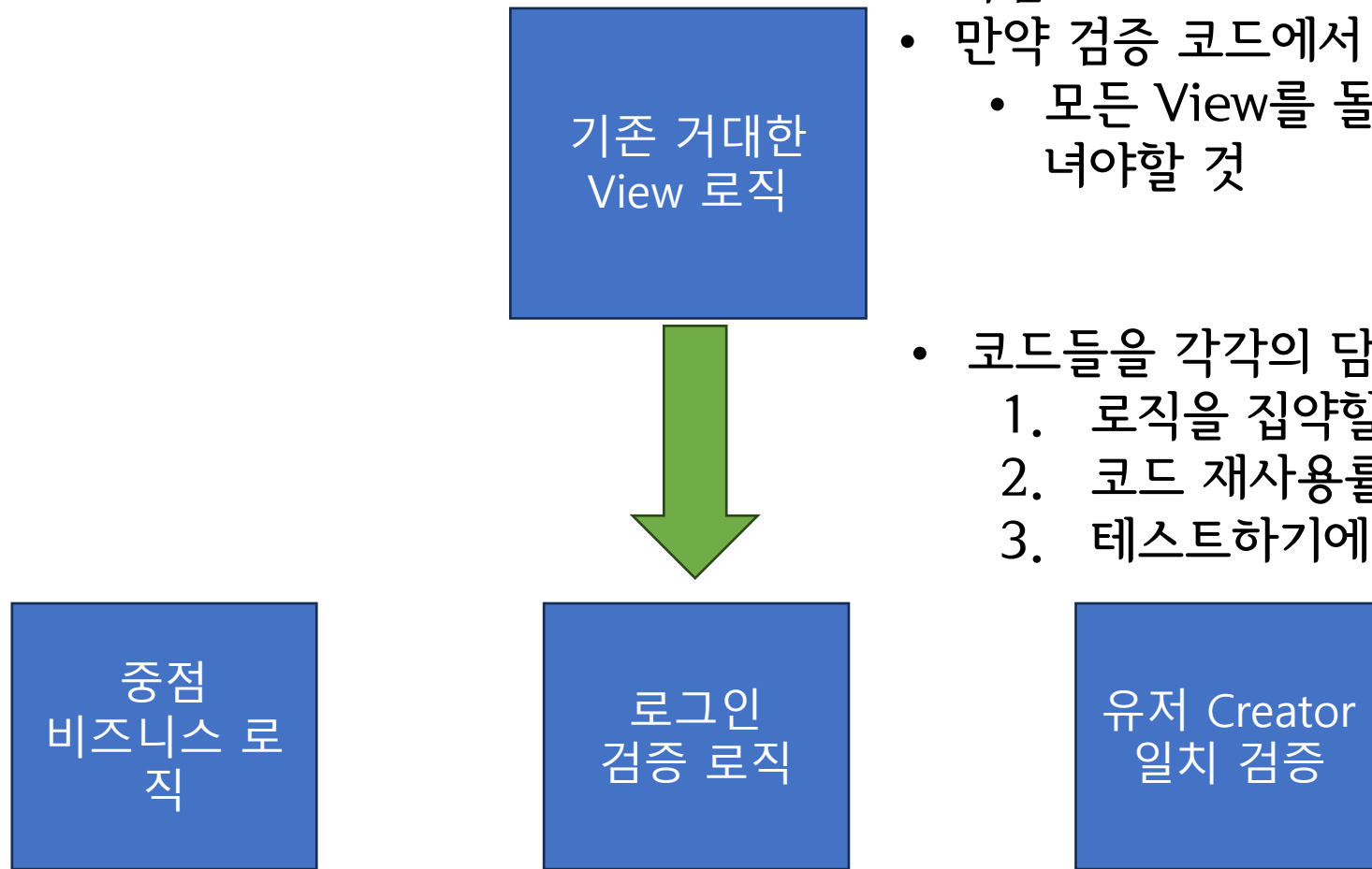
글이 본인 거인지 확인하기

작성자를 확인하는 로직, 로그인 확인 로직은 게시글 뿐만 아니라 댓글에도 적용되는 로직입니다. 또한, 다른 서비스가 늘어남에 따라 적용될 분야가 넓어질 수 있습니다.

즉, 해당 로직은 게시물 삭제, 생성 등의 메인 비즈니스 로직으로부터 분리된채로 존재해야, 관심사의 분리를 이루어낼 수 있고, 유지보수의 이점을 얻을 수 있습니다.

# Django.contrib.auth

직관적 구조 : 관심사의 분리



- 기존의 거대한 View 로직에서는 비즈니스 로직에 집중하기 보다는 다른 검증 로직 등이 추가되며 코드가 길어지고 중복됨
- 만약 검증 코드에서 수정 사항이 발견되면?
  - 모든 View를 돌아다니면서 중복 코드를 수정하고 다녀야할 것
- 코드들을 각각의 담당 로직에 따라 분리한다면
  1. 로직을 집약할 수 있어 이해도가 올라가고,
  2. 코드 재사용률을 높혀, 중복된 코드를 줄일 수 있음
  3. 테스트하기에 용이함

# 백엔드 구현 : 검증 로직 추가 위치 선정

메인 로직

로그인하는지 체크하기

글이 본인 거인지 확인하기

```
def edit(request, post_pk):
    post = Post.objects.get(pk=post_pk)

    if request.method == 'POST':
        title = request.POST['title']
        content = request.POST['content']
        Post.objects.filter(pk=post_pk).update(
            title=title,
            content=content
        )
        return redirect('detail', post_pk)

    return render(request, 'edit.html', {'post': post})

def delete(request, post_pk):
    post = Post.objects.get(pk=post_pk)
    post.delete()
    return redirect('home')
```

최대한 관심사의 분리를  
얻기 위해서는 로직을 어  
떻게 추가할까?

함수로 분리?

앞에서 이미 사실 스포를  
했죠? Decorator를 쓸  
것입니다.

# Django.contrib.auth

## 코드의 효율적 구조 - Decorator

### decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for `decorators` are `classmethod()` and `staticmethod()`.

The `decorator` syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):  
    ...  
f = staticmethod(f)  
  
@staticmethod  
def f(arg):  
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for [function definitions](#) and [class definitions](#) for more about `decorators`.

# 백엔드 구현 : 검증 로직 추가 위치 선정

```
@login_required
@check_is_owner_or_admin(Post, "post_pk")
def edit(request, post_pk):
    post = Post.objects.get(pk=post_pk)

    if request.method == "POST":
        title = request.POST["title"]
        content = request.POST["content"]
        Post.objects.filter(pk=post_pk).update(title=title, content=content)
        return redirect("detail", post_pk)

    return render(request, "edit.html", {"post": post})
```

Login\_required decorator는 django에서 제공하는 decorator(annotation)으로 로그인된 유저만 해당 허용하는 검증 로직을 추가하는 것입니다.

If user.is\_authenticated: 와 같은 로직입니다. 그냥 멋있게 적은 것예요

그럼...check\_is\_owner\_or\_admin은 무슨 로직일까요?



# 백엔드 구현 : 검증 로직 추가 위치 선정

```
def is_owner_or_admin(request, obj):
    return obj.creator == request.user or request.user.is_superuser

def check_is_owner_or_admin(model_cls, lookup_field="pk"):
    def decorator(view_func):
        @wraps(view_func)
        def _wrapped_view(request, *args, **kwargs):
            # Assume the object's ID is passed as 'pk' in kwargs
            obj_id = kwargs.get(lookup_field)
            if not obj_id:
                return render(request, "error.html", {"error": "Object ID not found."})

            # Retrieve the object based on ID and your model
            # You might need to adjust this part based on how your models are structured
            obj = get_object_or_404(model_cls, **{"pk": obj_id})

            # Call the permission check function
            if not is_owner_or_admin(request, obj):
                return render(request, "error.html", {"error": "Permission denied."})

            return view_func(request, *args, **kwargs)

        return _wrapped_view

    return decorator
```

함수 안에 함수가 있어..?!

심지어 그 함수도 view\_func이라는 함수를 인자로 가지고 실행한 값을 인자로 받아..?

함수 -> 함수 -> 함수???

HINT:

View\_func은 decorator가 감싸고 있는 views.py의 함수를 말합니다.

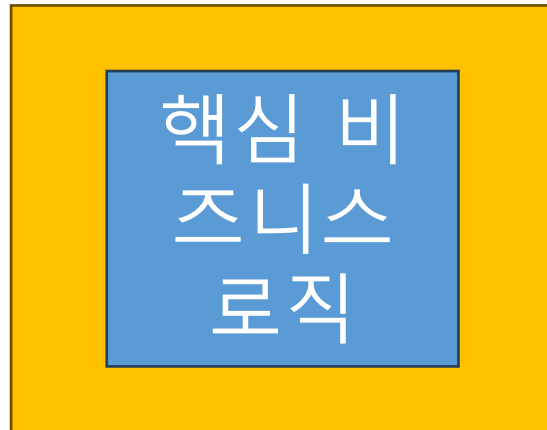
즉 wrapped\_view는 무언가를 실행하고, 그 다음 view\_func을 실행하고, 그 결과 값을 리턴하는 함수이네요

# 백엔드 구현 : 검증 로직 추가 위치 선정

Decorator도 결국은 함수였네요!

Decorator의 기능은 결국 래핑 당하는 함수(`views_func`)을 대체하기 위한 새로운 함수를 리턴하는 함수였습니다.

즉, 해당 `view_func`을 (새로운 기능 + `view_func`의 기능)을 수행하는 새로운 함수로 바꿔치기했다는 뜻입니다.



# 백엔드 구현 : check\_is\_creator\_or\_admin

```
def is_owner_or_admin(request, obj):
    return obj.creator == request.user or request.user.is_superuser

def check_is_owner_or_admin(model_cls, lookup_field="pk"):
    def decorator(view_func):
        @wraps(view_func)
        def _wrapped_view(request, *args, **kwargs):
            # Assume the object's ID is passed as 'pk' in kwargs
            obj_id = kwargs.get(lookup_field)
            if not obj_id:
                return render(request, "error.html", {"error": "Object ID not found."})

            # Retrieve the object based on ID and your model
            # You might need to adjust this part based on how your models are structured
            obj = get_object_or_404(model_cls, **{"pk": obj_id})

            # Call the permission check function
            if not is_owner_or_admin(request, obj):
                return render(request, "error.html", {"error": "Permission denied."})

            return view_func(request, *args, **kwargs)

        return _wrapped_view

    return decorator
```

결국 이 decorator는 검증해야할 모델 클래스와, 찾아야하는 인자를 입력받고, view\_func이 실행되기 전에

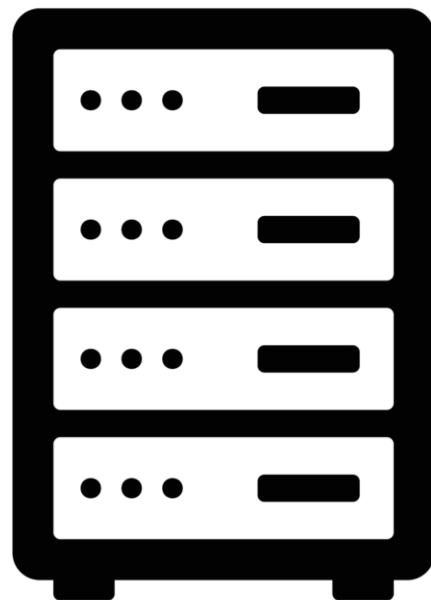
1. 해당 인자를 pk로 하는 객체가 존재하는지 확인
2. 해당 객체의 작성자가 view를 요청하는 유저인지 확인

이 두가지의 검증을 거친 다음에 view\_func을 수행하는 데코레이터입니다.

# 백엔드 구현 : check\_is\_owner\_or\_admin : 예시



Blog/edit/<post\_pk>로 POST 요청



@login\_required  
Session data 기반으로 유저 인증



@check\_is\_creator\_or\_admin  
인증된 유저 정보를 기반으로 작성자 또는 어드민인지 확인



원래 목표였던 VIEW function 실행

# 백엔드 구현 : check\_is\_owner\_or\_admin : 예시



@login\_required



# | 백엔드 구현: 실제 예시

서버 열어서 보여드릴게요

# 오늘 배운 것 복기

## 1. 백엔드의 설계 과정

- 요구사항 정의 : 무엇을 만들어야 하는가?
- 이벤트 스토밍 : 어떤 객체들이 어떤 행동으로 어떻게 변화하지?
- 도메인 구체화 : 연관 있는 객체들과 그 행동을 묶고 다른 묶음들과 구별하기
- 시퀀스 다이어그램 : 어떤 플로우로 로직이 진행되지? (라이프 사이클)
- 실제 개발 시작
- <https://incheol-jung.gitbook.io/docs/q-and-a/architecture/ddd>
- 꼭 이렇게 설계할 필요는 없어요! 나중에 프로젝트가 복잡해질 때 진가를 발휘합니다.
- 키워드: Domain-Driven-Design, 3 layer architecture, Hexagonal

# | 오늘 배운 것 복기

## 2. Django의 보안 구현

- Django는 django.auth에서 보안 관련 정말 많은 것들을 이미 구현된 채로 제공합니다.
- AbstractUser > AbstractBaseUser
- Login\_required
- ModelForm
- Session Data(login, logout)
  
- 자잘한 것들은 물론 직접 구현해야 합니다.
- Check\_is\_creator\_or\_admin



# 과제

## 오늘의 과제: 프로젝트 만들기

1. 오늘 배운 내용 + 제공해드린 코드 + GPT + 인터넷을 이용해서 과제를 수행해 주세요!
  - 남의 코드를 사용하는 것도 능력! 물론, 여러분들이 코드를 이해하고 이용할 수 있도록 과제가 나갑니다
  - 질문도 물론 허용, 협업 완전 장려
  - 다만 진짜로 남의 과제 완성본 자체를 베끼지는 말아주세요(GPT 짜깁기는 가능해요 하지만...굳이..?)
2. 많이 어렵고 오래 걸릴 수도 있지만, django documentation, 블로그 글, stack overflow 등을 읽으면서 코드를 만들면 정말 성장하실 수 있을 거예요. 오히려 재미있을 수도..?
3. 디자인 신경 안 쓰셔도 괜찮습니다. 그냥 순수 html로만 하셔도 돼요.

# 과제

## 오늘의 과제: 프로젝트 만들기

### 1. 블로그 만들기!

- 글과 댓글을 작성할 수 있는 블로그를 만들어주세요
- 작성자만 글, 댓글을 삭제할 수 있도록 해주세요 (User가 필요하겠죠)?
- 로그인한 사람들만 글의 세부 내용을 알 수 있도록 해주세요
- 글의 세부 내용을 볼 때, 작성자가 누구인지 이름을 추가해 주세요
- 글의 세부 내용을 보여줄 때, 마지막으로 열람된 시간, 마지막으로 열람한 유저 이름을 같이 볼 수 있도록 해주세요.

### 2. 마지막으로 열람한 시간, 유저

- 마지막으로 열람한 시간과 유저를 수정하는 로직은 decorator로 제작해 주세요
- Hint: decorator를 유저가 세부 내용을 확인하는 로직 함수 위에 추가해야겠죠?
- Hint: 열람한 사람과 시간을 저장하려면 **저장하는 인자가** 있어야겠죠?

# 과제

## 오늘의 과제: 프로젝트 만들기

### 3. 회원가입

- 유저가 회원가입 할 수 있는 form을 django form.ModelForm을 이용해서 만들어주세요!
- 유저의 아이디와 비밀번호 모두 8자를 넘는지를 검증해주세요!
- Hint: django의 is\_valid가 실행하는 clean\_<custom form> 사용해주세요!

# 과제

## 오늘의 과제: 프로젝트 만들기 : TRY

### 1. 유저 개인 페이지 만들기

- 유저가 쓴 모든 글을 확인할 수 있는 개인 페이지를 만들어주세요!
- Navbar에서 클릭하면 바로 로그인된 유저의 개인 페이지로 갈 수 있는 버튼을 만들어주세요
  - 로그인 되어 있을 때만 보여야해요!
- 유저 개인 페이지에서는 유저가 구독한 다른 유저가 누구인지 볼 수 있도록 해주세요
- 유저 개인 페이지에서 그 유저를 구독할 수 있는 버튼을 만들어주세요!
  - 누르면 home이든 다시 그 유저 페이지로 가게 해주세요
- 구독 버튼 글 세부내용 확인할 때도 확인 가능하게 해주세요
- 세부 내용에서 작성자 이름을 누르면 해당 작성자의 개인 페이지로 갈 수 있게 해주세요

끝: 고생하셨습니다!!!!

