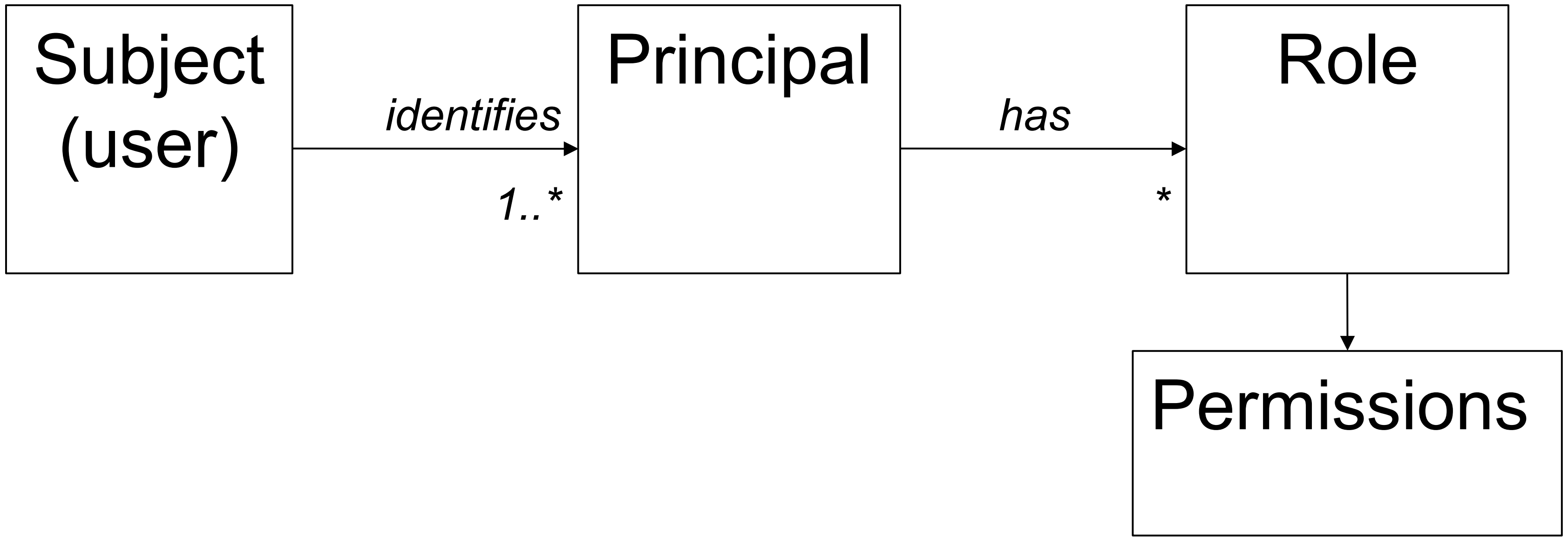
Authentication in Django

Role Based Authorization

Permissions are based on the *roles* a user possesses.

A user may have many roles.

Example: “joe” has roles “voter” and “administrator”

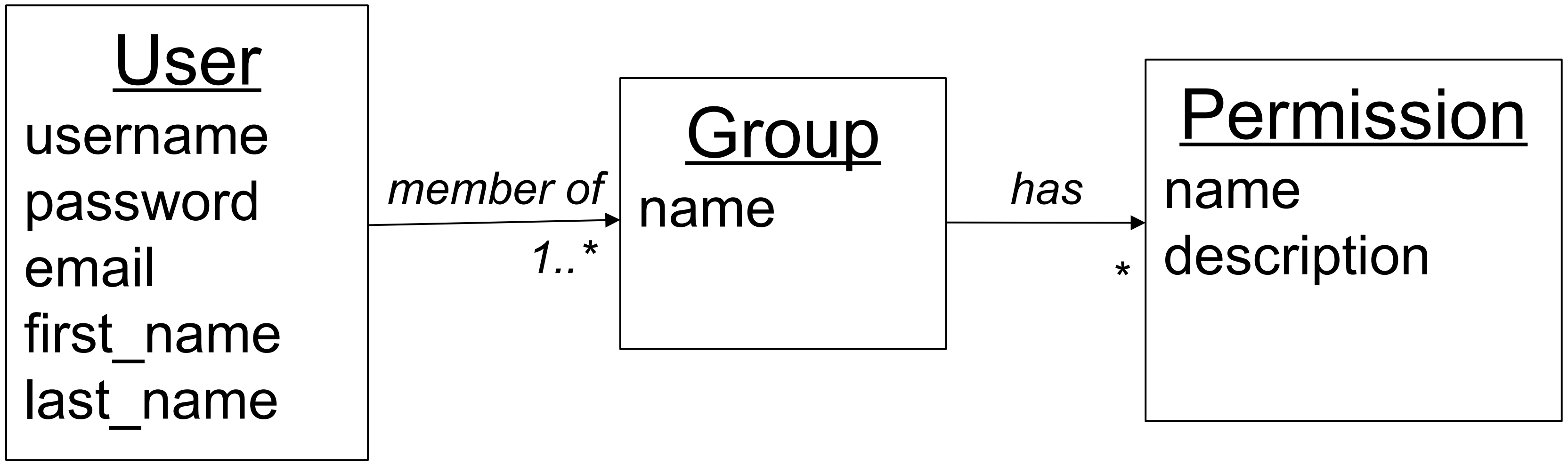


How Django Does It

User - identifies a user, authenticate using one of many *backends.*

Group - User is assigned one or more groups. Each group has some Permissions.

Permission - key-value pair (anything you like) used in code to enforce *authorization*



Checking Authorization in Code

```
from django.contrib.auth
           import authenticate, login

def dumb_login_view(request):
  # authenticate first
  user = authenticate(request, "hacker", "Hack!")
  login(request, user)

  if user.is_authenticated:
     # allow any logged in user to do something

  if user.has_perm('blog.can_create'):
     # allow user to create a blog entry
```

Checking Auth in Views

The `request` object has reference to current user.

```
def blog_index(request):
    if not request.user.is_authenticated:
        return redirect('login')

    if request.user.has_perm(
                'blog.can_post_comment')
        # include form for posting comments
```

Use Decorators on Views

Decorators reduce risk of errors, create cleaner code

```
from django.contrib.auth.decorators
   import login_required, permission_required

@login_required
def blog_index(request):
    """show index of todos for this user"""

@permission_required('blog.can_create')
def add(request):
    """post a new blog entry"""
```

Decorators in urls.py

You can add decorators in urls.py. I think using decorators in views is more readable & avoids errors.

```
urlpatterns = [

   path('blog/', login_required(views.index)),
   ...
```

Define Your Own Decorators

If none of Django's decorators do what you want...

https://docs.djangoproject.com/en/3.0/topics/auth/default/

```
def kasetsart_email(user) -> bool:
    return user.email.endswith('@ku.ac.th')

@user_passes_test( kasetsart_email )
def vote(request, question_id):
    # only users at KU can vote
```

Mixins for Class-based Views

"Mixin" means to combine or "mix in" behavior from several different classes.

```
from django.contrib.auth.mixins
            import LoginRequiredMixin

class PollsIndex(LoginRequiredMixin, ListView):
    template_name = 'polls/index.html'
    ...
```

Authorization Checks in Templates

Templates can use the **`user`** and **`perms`** objects.

```
{% if user.is_authenticated %}
    Hello, {{ user.username }}
{% else %}
    Please <a href="{% url 'login'%}">Login</a>
{% endif %}

{# same as user.has_perm('blog.post_entry') #}
{% if perms.blog.post_entry %}
   You can post a blog entry
{% endif %}
```

Where to Apply Authorization?

1. In templates. Gives web page the desired appearance and page flow, but can be by-passed. Don't rely on it.

2. In views. Requests are always passed to a view, so this is fairly secure. Prefer decorators or Mixins instead of checks in code.

3. In models? In some frameworks, you can configure required permissions directly into model classes. Apparently not in Django.

4. In url mapping (urls.py).

Using OAuth & OpenId

Use the **`django-allauth`** package.

Both django-allauth and django-social-auth extensions add OAuth support to Django, but django-allauth also manages local accounts, simplifying your code.